A Lightweight Infrastructure for Graph Analytics*

Donald Nguyen, Andrew Lenharth and Keshav Pingali
The University of Texas at Austin, Texas, USA
{ddn@cs, lenharth@ices, pingali@cs}.utexas.edu

Abstract

Several domain-specific languages (DSLs) for parallel graph analytics have been proposed recently. In this paper, we argue that existing DSLs can be implemented on top of a general-purpose infrastructure that (i) supports very fine-grain tasks, (ii) implements autonomous, speculative execution of these tasks, and (iii) allows application-specific control of task scheduling policies. To support this claim, we describe such an implementation called the Galois system.

We demonstrate the capabilities of this infrastructure in three ways. First, we implement more sophisticated algorithms for some of the graph analytics problems tackled by previous DSLs and show that end-to-end performance can be improved by orders of magnitude even on power-law graphs, thanks to the better algorithms facilitated by a more general programming model. Second, we show that, even when an algorithm can be expressed in existing DSLs, the implementation of that algorithm in the more general system can be orders of magnitude faster when the input graphs are road networks and similar graphs with high diameter, thanks to more sophisticated scheduling. Third, we implement the APIs of three existing graph DSLs on top of the common infrastructure in a few hundred lines of code and show that even for power-law graphs, the performance of the resulting implementations often exceeds that of the original DSL systems, thanks to the lightweight infrastructure.

*This research was supported by NSF grants CCF 1337281, CCF 1218568, ACI 1216701, and CNS 1064956. Donald Nguyen was supported by a DOE Sandia Fellowship. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant OCI-1053575.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522739

1 Introduction

Graph analysis is an emerging and important application area. In many problem domains that require graph analysis, the graphs can be very large; for example, social networks today can have a billion nodes. Parallel processing is one way to speed up the analysis of such large graphs, but writing efficient parallel programs, especially for shared-memory machines, can be difficult for programmers.

Several domain-specific languages (DSLs) for graph analytics have been proposed recently for simplifying the task of writing these programs [11, 12, 17–19, 31]. Programs are expressed as iterated application of *vertex operators*, where a vertex operator is a function that reads and writes a node and its immediate neighbors. Parallelism is exploited by applying the operator to multiple nodes of the graph simultaneously in rounds in a bulk-synchronous style; *coordinated scheduling* inserts the necessary synchronization to ensure that all the operations in one round finish before the next one begins.

In this paper, we argue that this programming model is insufficient for high-performance, general-purpose graph analytics where, by general-purpose, we mean diversity both in algorithms and in the types of input graphs being analyzed. We show that to obtain high performance, some algorithms require *autonomous scheduling*, in which operator applications are scheduled for execution whenever their data becomes available. For efficiency, autonomous scheduling requires application-specific priority functions that must be provided by programmers and must therefore be supported by the programming model. We also identify domain-specific optimizations that must be supported by such systems.

Since these features are properties of programming models and not their implementations, we argue that continued improvement of DSLs that *only* provide for coordinated execution of vertex programs is inadequate to meet the challenges of graph analytics.

To support this argument, this paper introduces several improvements to an existing system, **Galois** [15], that provide for a rich programming model with coordinated and autonomous scheduling, and with and without

application-defined priorities. The main improvements are: a topology-aware work-stealing scheduler, a priority scheduler, and a library of scalable data structures.

We demonstrate the capabilities of the improved Galois infrastructure in three ways. First, we implement more sophisticated algorithms for some of the graph analytics problems tackled by previous DSLs and show that end-to-end performance can be improved by many orders of magnitude even on power-law graphs, thanks to better algorithms. Second, we show that even when an algorithm can be expressed in existing DSLs, the implementation of that algorithm in the more general system can be orders of magnitude faster when the input graphs are road networks and similar graphs with high diameter, thanks to more sophisticated scheduling policies. Third, we implement the APIs of three existing graph DSLs on top of the common infrastructure in a few hundred lines of code and show that, even for power-law graphs, the performance of the resulting implementations often exceeds that of the original DSL systems, thanks to the lightweight infrastructure.

The rest of this paper is organized as follows. In Section 2, we describe the programming models and DSLs under consideration. In Section 3, we describe several graph analytics problems as well as algorithms for solving them. In Section 4, we describe key improvements to the Galois system. We evaluate the DSL programming models and their implementations in Section 5. Lessons and conclusions are presented in Section 6.

2 Programming models and DSLs

Many graph analysis algorithms can be written as iterative programs in which each iteration updates labels on nodes and edges using elementary graph operations. In this section, we describe abstract programming models for such algorithms and describe how these models are implemented in four graph DSLs.

2.1 Model problem: SSSP

Given a weighted graph $G = (V, E, w)$, where V is the set of nodes, E is the set of edges, and w is a map from edges to edge weights, the single-source shortest-paths (**SSSP**) problem is to compute the distance of the shortest path from a given source node s to each node. Edge weights can be negative, but it is assumed that there are no negative weight cycles.

In most SSSP algorithms, each node is given a label that holds the distance of the shortest known path from the source to that node. This label, which we call $dist(v)$, is initialized to 0 for s and ∞ for all other nodes. The basic SSSP operation is *relaxation* [10]: given an edge

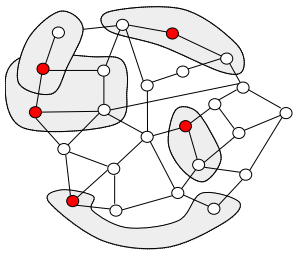


Figure 1: Amorphous data-parallelism programming model.

	Cycles	Inst.
bfs	6007	2077
sssp	1521	308
dia	7265	2296
cc	5063	1380
pr	3190	541

Figure 2: Cycle and instruction counts of operators for several applications (application details §3; measurement details §5.1).

(u, v) such that $dist(u) + w(u, v) < dist(v)$, the value of $dist(v)$ is updated to $dist(u) + w(u, v)$. Each relaxation, therefore, lowers the *dist* label of a node, and when no further relaxations can be performed, the resulting node labels are the shortest distances from the source to the nodes, regardless of the order in which the relaxations were performed.

Nevertheless, some relaxation orders may converge faster and are therefore more work-efficient than others. For example, Dijkstra's SSSP algorithm [10] relaxes each node just once by using the following strategy: from the set of nodes that have not yet been relaxed, pick one that has the minimal label.

However, Dijkstra's algorithm does not have much parallelism, so some parallel implementations of SSSP use this rule only as a heuristic for priority scheduling: given a choice between two nodes with different *dist* labels, they pick the one with the smaller label, but they may also execute some nodes out of priority order to exploit parallelism. The price of this additional parallelism is that some nodes may be relaxed repeatedly. A balance must be struck between controlling the amount of extra work and exposing parallelism.

2.2 Programming models

To discuss common issues in graph analytics, it is convenient to use the terminology of *amorphous data-parallelism* (ADP) [26], a data-centric programming model for expressing parallelism in regular and irregular algorithms. The basic concepts of ADP are illustrated in Figure 1. *Active nodes* are nodes in the graph where computation must be performed; they are shown as filled dots in Figure 1. The computation at an active node is called an *activity*, and it results from the application of an *operator* to the active node.[1] The operator is a composition

[1] In some algorithms, it is more convenient to think in terms of active *edges* rather than active nodes. Without loss of generality, in this paper, we will use the term active nodes.

of elementary graph operations with other arithmetic and logical operations. The set of graph elements read and written by an activity is its *neighborhood*. The neighborhood of the activity at each active node in Figure 1 is shown as a "cloud" surrounding that node. If there are several data structures in an algorithm, neighborhoods may span multiple data structures. In general, neighborhoods are distinct from the set of immediate neighbors of the active node, and neighborhoods of different activities may overlap. In a parallel implementation, the semantics of reads and writes to such overlapping regions must be specified carefully.

The key design choices when writing parallel graph analytics programs can be summarized as follows: *what* does the operator do, *where* in the graph is it applied, and *when* is the corresponding activity executed.

What does the operator do? In general, the operator expresses some computation on the neighborhood elements, and it is allowed to *morph* [26] the graph structure of the neighborhood by adding or removing nodes and edges. Graph analytics applications typically require simpler operators that only update labels on neighborhood nodes and edges, keeping the graph structure invariant, so we focus on these *local computation* operators in this paper.

In some graph problems such as SSSP, operators can be implemented in two general ways that we call *push* style or *pull* style. A push-style operator reads the label of the active node and writes to the labels of its neighbors; information flows from the active node to its neighbors. A push-style SSSP operator attempts to update the *dist* label of the immediate neighbors of the active node by performing relaxations with them. In contrast, a pull-style operator writes to the label of the active node and reads the labels of its neighbors; information flows to the active node from its neighbors. A pull-style SSSP operator attempts to update the *dist* label of the active node by performing relaxations with each neighbor of the active node. In a parallel implementation, pull-style operators require less synchronization since there is only one writer per active node.

Figure 2 shows the average numbers of cycles and executed instructions for the operators (including runtime overhead) of the graph analytics applications discussed in this paper. We see that *graph analytics algorithms use very fine-grain operators that may execute only a few hundred instructions*.

Where is the operator applied? Implementations can be *topology-driven* or *data-driven*.

In a *topology-driven* computation, active nodes are defined structurally in the graph, and they are independent of the values on the nodes and edges of the graph. The Bellman-Ford SSSP algorithm is an example [10]; this algorithm performs $|V|$ supersteps, each of which applies a push-style or pull-style operator to all the edges. Practical implementations terminate the execution if a superstep does not change the label of any node. Topology-driven computations are easy to parallelize by partitioning the nodes of the graph between processing elements.

In a *data-driven* computation, nodes become active in an unpredictable, dynamic manner based on data values, so active nodes are maintained in a worklist. In a data-driven SSSP program, only the source node is active initially. When the label of a node is updated, the node is added to the worklist if the operator is push-style; for a pull-style operator, the neighbors of that node are added to the worklist. Data-driven implementations can be more work-efficient than topology-driven ones since work is performed only where it is needed in the graph. However, load-balancing is more challenging, and careful attention must be paid to the design of the worklist to ensure it does not become a bottleneck.

When is an activity executed? When there are more active nodes than threads, the implementation must decide which active nodes are prioritized for execution and when the side-effects of the resulting activities become visible to other activities. There are two popular models that we call *autonomous scheduling* and *coordinated scheduling*.

In autonomous scheduling, activities are executed with transactional semantics, so their execution appears to be atomic and isolated. Parallel activities are serializable, so the output of the overall program is the same as some sequential interleaving of activities. Threads retrieve active nodes from the worklist and execute the corresponding activities, synchronizing with other threads only as needed to ensure transactional semantics. This fine-grain synchronization can be implemented using speculative execution with logical locks or lock-free operations on graph elements. The side-effects of an activity become visible externally when the activity commits.

Coordinated scheduling, on the other hand, restricts the scheduling of activities to rounds of execution, as in the Bulk-Synchronous Parallel (BSP) model [33]. The execution of the entire program is divided into a sequence of supersteps separated by barrier synchronization. In each superstep, a subset of the active nodes is selected and executed. Writes to shared-memory, in shared-memory implementations, or messages, in distributed-memory implementations, are considered to be communication from one superstep to the following superstep. Therefore, each superstep consists of updating memory based on communication from the previous superstep, performing computations, and then issuing communication to the next superstep. Multiple updates to the

same location are resolved in different ways as is done is the varieties of PRAM models, such as by using a reduction operation [14].

Application-specific priorities Of the different algorithm classes discussed above, data-driven, autonomously scheduled algorithms are the most difficult to implement efficiently. However, they can converge much faster than algorithms that use coordinated scheduling [3]. Moreover, for high-diameter graphs like road networks, data-driven autonomously scheduled algorithms may be able to exploit more parallelism than algorithms in other classes; for example, in BFS, if the graph is long and skinny, the number of nodes at each level will be quite small, limiting parallelism if coordinated scheduling is used.

Autonomously scheduled, data-driven graph analytics algorithms require application-specific priorities and priority scheduling to balance work-efficiency and parallelism.

One example is delta-stepping SSSP [21], the most commonly used parallel implementation of SSSP. The worklist of active nodes is implemented, conceptually, as a sequence of bags, and an active node with label d is mapped to the bag at position $\lfloor \frac{d}{\Delta} \rfloor$, where Δ is a user-defined parameter. Idle threads pick work from the lowest-numbered non-empty bag, but active nodes within the same bag may execute in any order relative to each other. The optimal value of Δ depends on the graph.

The general picture is the following. Each task t is associated with an integer *priority(t)*, which is a heuristic measure of the importance of that task for early execution relative to other tasks. For delta-stepping SSSP, the priority of an SSSP relaxation task is the value $\lfloor \frac{d}{\Delta} \rfloor$. We will say that a task t_1 has *earlier priority* than a task t_2 if $priority(t_1) < priority(t_2)$. It is permissible to execute tasks out of priority order, but this may possibly lower work efficiency. A good DSL for graph applications must permit application programmers to specify such application and input-specific priorities for tasks, and the runtime system must schedule these fine-grain tasks with minimal overhead and minimize priority inversions.

2.3 Graph analytics DSLs

Graph analytics DSLs usually constrain programmers to use a subset of features described in Section 2.2.

GraphLab [18] is a shared-memory programming model for topology or data-driven computations with autonomous or coordinated scheduling, but it is restricted to *vertex programs*. A vertex program has a graph operator that can only read from and write to the immediate neighbors of the active node. There are several priority scheduling policies available, but the implementation of the priority scheduler is very different from the one used in the Galois system (§4.1.2).

	bfs	cc	dia	pr	sssp	bc
Galois	C+A	A	C+A	C	A	A
GraphLab	A	A	A	C	A	
Ligra	C	C	C	C	C	C
PowerGraph	C	C	C	C	C	
GraphChi		C		C		

Figure 3: Summary of whether an application uses coordinated (C) or autonomous (A) scheduling.

PowerGraph [11] is a distributed-memory programming model for topology or data-driven computations with autonomous or coordinated scheduling, but it is restricted to *gather-apply-scatter* (GAS) programs, which are a subset of vertex programs. Graphs are partitioned by edge where the endpoints of edges may be shared by multiple machines. Values on shared nodes can be resolved with local update and distributed reduction. On scale-free graphs, which have many high-degree nodes, this is a useful optimization to improve load balancing. PowerGraph supports autonomous scheduling, but the scheduling policy is fixed by the system and users cannot choose among autonomous policies.

GraphChi [17] is a shared-memory programming model for vertex programs that supports out-of-core processing when the input graph is too large to fit in memory. GraphChi relies on a particular sorting of graph edges in order to provide IO-efficient access to both the in and out edges of a node. Since computation is driven by the loading and storing of graph files, GraphChi only provides coordinated scheduling.[2]

Ligra [31] is a shared-memory programming model for vertex programs with coordinated scheduling. A unique feature of Ligra is that it switches between push and pull-based operators automatically based on a user-provided threshold.

3 Applications

As we argued in Section 2, graph analytics algorithms can use rich programming models, but most existing graph DSLs support only a simple set of features. In this section, we describe how six graph analytics problems are solved in five different systems in light of these programming model restrictions. Most of the applications are provided by the DSL systems themselves, except for

[2] GraphChi takes a program that could be autonomously scheduled but imposes a coordinated schedule for execution.

GraphLab, for which we implemented the applications ourselves. Figure 3 summarizes applications based on one dimension introduced in Section 2: whether they use autonomous or coordinated scheduling.

Single-source shortest-paths The best sequential and parallel algorithms for SSSP use priority scheduling. The Galois SSSP application uses the data-driven, autonomously scheduled delta-stepping algorithm (§2), using auto-tuning to find an optimal value of Δ for a given input. GraphLab does not provide an SSSP application, so we created one based on the Galois application, using the priority scheduling available in GraphLab. SSSP can be solved without using priorities by applying data-driven execution of the Bellman-Ford algorithm (§2). Since PowerGraph and Ligra do not support priority scheduling, they both provide SSSP applications based on this method.

Breadth-first search Breadth-first search (**BFS**) numbering is a special case of the SSSP problem in which all edge weights are one. Depending on the structure of the graph, there are two important optimizations. For low-diameter graphs, it beneficial to switch between push and pull-based operators, which reduces the total number of memory accesses. For high-diameter graphs, it is beneficial to use autonomous scheduling. Coordinated execution with high-diameter graphs produces many rounds with very few operations per round, while autonomous execution can exploit parallelism among rounds.

The Galois BFS application blends coordinated and autonomous scheduling. Initially, the application uses coordinated scheduling of the push and pull-based operators. After a certain number of rounds of push-based traversals, the application switches to prioritized autonomous scheduling. The priority function favors executing nodes with smaller BFS numbers.

The Ligra application uses coordinated scheduling and switches between push-based and pull-based operators automatically. Since PowerGraph does not provide a BFS application, we created one based on its SSSP application. GraphLab does not provide a BFS application, so we created one based on prioritized autonomous scheduling.

Approximate diameter The diameter of a graph is the maximum length of the shortest paths between all pairs of nodes. The cost of computing this exactly is prohibitive for any large graph, so many applications call for an approximation of the diameter (**DIA**) of a graph. In this paper, we limit ourselves to computing the diameter of unweighted graphs.

The Galois application is based on finding pseudo-peripheral nodes in the graph. It begins by computing a BFS from an arbitrary node. Then, it computes another BFS from the node with maximum distance, discovered by the first BFS. In the case of ties for maximum distance, the algorithm picks a node with the least degree. It continues this process until the maximum distance does not increase. Each BFS is an instance of the blended algorithm described above.

GraphLab does not provide an application for this problem; we created one based on the pseudo-peripheral algorithm. Ligra provides an approximate diameter application that uses the coordinated execution of BFS from k starting nodes at the same time. The k parameter is chosen such that the search data for a node fits in a single machine word. PowerGraph includes an approximate diameter application based on probabilistic counting, which is used to estimate the number of unique vertex pairs with paths with a distance at most k. When the estimate converges, k is an estimation of the diameter of the graph.

Betweenness centrality Betweenness centrality (**BC**) is a measure of the importance of a node in a graph. A popular algorithm by Brandes [4] computes the betweenness centrality of all nodes by using forward and backward breadth-first graph traversals. The Galois application is based on a priority-scheduled, pull-based algorithm for computing betweenness centrality. The priority function is based on the BFS number of a node. The Ligra application switches between pull and push-based operators with coordinated execution, which can have significant overhead on large diameter graphs.

Connected components In an undirected graph, a connected component (**CC**) is a maximal set of nodes that are reachable from each other.

Galois provides a parallel connected components application based on a concurrent union-find data structure. It is a topology-driven computation where each edge of the graph is visited once to add it to the union-find data structure.

PowerGraph, GraphChi and Ligra include applications based on iterative label propagation. Each node of the graph is initially given a unique id. Then, each node updates its label to be the minimum value id among itself and its neighbors. This process continues until no node updates its label; it will converge slowly if the diameter of the graph is high.

GraphLab does not provide an algorithm for finding connected components; we implemented one based on the label propagation algorithm.

PageRank PageRank (**PR**) is an algorithm for computing the importance of nodes in an unweighted graph.

GraphLab, GraphChi, PowerGraph and Ligra have two coordinated push-based applications, which are either topology-driven or data-driven. We use the topology-driven application in all cases. A possible priority function is to prefer earlier execution of nodes with the greatest change in value; although none of the PageRank applications evaluated use this function.

GraphChi has both a vertex program application as well as a gather-apply-scatter application. We use the latter because it is slightly faster.

Galois provides a pull-based PageRank application that reduces the memory overhead and synchronization compared to push-based applications.

4 The Galois system

The Galois system is an implementation of the amorphous data-parallelism (ADP) programming model presented in Section 2.2. Application programmers are given a sequential programming language without explicitly parallel programming constructs like threads and locks. Key features of the system are the following.

- Application programmers specify parallelism *implicitly* by using an unordered-set iterator [26] which iterates over a worklist of active nodes. The worklist is initialized with a set of active nodes before the iterator begins execution. The execution of a iteration can create new active nodes, and these are added to the worklist when that iteration completes execution.

- The body of the iterator is the implementation of the operator, and it is an imperative action that reads and writes global data structures. Iterations are required to be *cautious*: an iteration must read *all* elements in its neighborhood before it writes to *any* of them [26]. In our experience, this is not a significant restriction since the natural way of writing graph analytics applications results in cautious iterations.

- The relative order in which iterations are executed is left unspecified in the application code; the only requirement is that the final result should be identical to that obtained by executing the iterations sequentially in some order. An optional application-specific priority order for iterations can be specified with the iterator [23], and the implementation tries to respect this order when it schedules iterations (§4.1).

- The system exploits parallelism by executing iterations in parallel. To ensure serializability of iterations, programmers must use a library of built-in concurrent data structures for graphs, worklists, etc. (§4.2). These library routines expose a standard API to programmers, and they implement lightweight synchronization to ensure serializability of iterations, as explained below.

Inside the data structure library, the implementation of a data structure operation, such as reading a graph node or adding an edge between two nodes, acquires logical locks on nodes and edges before performing the operation. If the lock is already owned by another iteration, the iteration that invoked the operation is rolled back, releasing all its acquired locks, and is retried again later. Intuitively, the cautiousness of iterations reduces the synchronization problem to the dining philosopher's problem [7], obviating the need for more complex solutions like transactional memory.

Another useful optimization is when an operator only performs a simple update to machine word or when transactional execution is not needed at all. For these cases, all data structure methods have an optional parameter that indicates whether an operation always or never acquires locks. Experienced users can disable locking and use machine atomic instructions if desired.

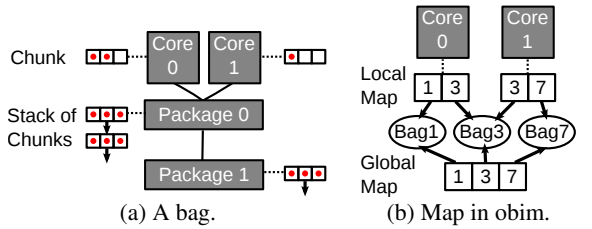
An accumulating collection is a collection of elements that supports concurrent insertion of new elements but does not need to support concurrent reads of the collection. Coordinated scheduling policies can be built from multiple autonomously scheduled loops and an accumulating collection data structure, which is provided by the Galois library (§4.2). For example, loop 1 executes, populating a collection with work that should be done by loop 2. Then, loop 1 finishes, and loop 2 iterates over the collection generated by loop 1, and so on. Control logic can be placed between loops, allowing the expression of sophisticated coordinated strategies like the pull versus push optimization (§2).

4.1 Scheduler

The core Galois scheduler is aware of the machine topology, and is described in Section 4.1.1. Priority scheduling can be layered on top of this scheduler as described in Section 4.1.2.

4.1.1 Topology-aware bags of tasks

When there are no application-specific priorities, the Galois scheduler uses a *concurrent bag* to hold the set of pending tasks (active nodes). The bag (depicted in Figure 4a) allows concurrent insertion and retrieval of



(a) A bag. (b) Map in obim.

Figure 4: Organization of scheduling data structures.

unordered tasks and is implemented in a distributed, machine-topology-aware way as follows.

- Each core has a data structure called a *chunk*, which is a ring-buffer that can contain 8–64 tasks (size chosen at compile time). The ring-buffer is manipulated as a stack (LIFO)[3]: new tasks are pushed onto the ring buffer, and tasks are popped from it when the core needs work.
- Each package has a list of chunks. This list is manipulated in LIFO order.
- When the chunk associated with a core becomes full, it is moved to the package-level list.
- When the chunk associated with a core becomes empty, the core probes its package-level list to obtain a chunk. If the package-level list is also empty, the core probes the lists of other packages to find work. To reduce traffic on the inter-package connection network, only one hungry core hunts for work in other packages on behalf of all hungry cores in a package.

4.1.2 Priority scheduling

Priority scheduling is used extensively in operating systems, but relatively simple implementations suffice in that context because tasks are relatively coarse-grained: OS tasks may execute in tens or hundreds of milliseconds, whereas tasks in graph analytics take only microseconds to execute, as shown in Figure 2. Therefore, the overheads of priority scheduling in the OS context are masked by the execution time of tasks, which is not the case in graph analytics, so solutions from the operating systems area cannot be used here.

Another possibility is to use a concurrent priority queue, but we were unable to get acceptable levels of performance with lock-free skip-lists [29] and other approaches in the literature (these alternatives are described in more detail at the end of this section). In this section, we describe a machine-topology-aware, physically distributed data structure called *obim* that exploits the fact that priorities are "soft," so the scheduler is not required to follow them exactly.

[3]It is possible to manipulate the bag in FIFO-style as well, but we omit this possibility to keep the description simple.

Overview Unlike the basic scheduler of Section 4.1.1 which uses just a bag, the obim scheduler uses a *sequence* of bags, where each bag is associated with one priority level. Tasks in the same bag have identical priorities and can therefore be executed in any order; however, tasks in bags that are earlier in the sequence are scheduled preferentially over those in later bags. This is shown pictorially as the *Global Map* in Figure 4b. This map is sparse since it contains bags only at entries 1, 3 and 7. Threads work on tasks in bag 1 first; only if a thread does not find a task in bag 1 does it look for work in the next bag (bag 3). If a thread creates a task with some priority and the corresponding bag is not there in the global map, the thread allocates a new bag, updates the global map, and inserts the task into that bag.

The global map is a central data structure that is read and written by all threads. To prevent it from becoming a bottleneck and to reduce coherence traffic, each thread maintains a software-controlled lazy cache of the global map, as shown in Figure 4b. Each local map contains some portion of the global map that is known to that thread, but it is possible for a thread to update the global map without informing other threads.

The main challenge in the design of obim is getting threads to work on early priority work despite the distributed, lazy-cache design. This is accomplished as follows.

Implementation of global/local maps The thread-local map is implemented by a sorted, dynamically resizable array of pairs. Looking up a priority in the thread-local map is done using a binary search. Threads also maintain a version number representing the last version of the global map they synchronized with.

The global map is represented as a log-based structure which stores bag-priority pairs representing insert operations on the logical global map. Each logical insert operation updates the global version number.

Updating the map: When a thread cannot find a bag for a particular priority using only its local map, it must synchronize with the global map and possibly create a new mapping there. A thread replays the global log from the point of the thread's last synchronized version to the end of the current global log. This inserts all newly created mappings into the thread's local map. If the right mapping is still not found, the thread will acquire a write lock, replay the log again, and append a new mapping to the global log and its local map. Some care must be taken with the implementation of the global log to ensure that

the log can be appended in the presence of concurrent readers without requiring locks.

Pushing a task: A thread pushing a task uses its local map to find the correct bag into which to insert. Failing that, the thread updates its local map from the global map, as above, possibly creating a new mapping, and it uses the found or created bag for the push operation.

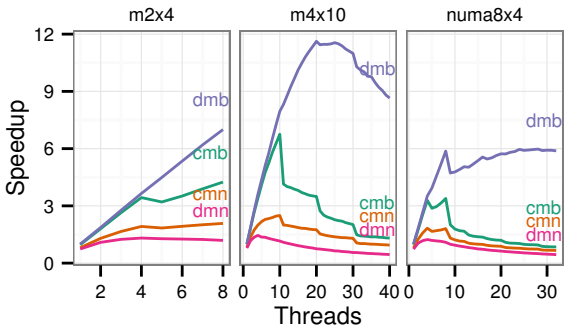
Retrieving a task: To keep close to the ideal schedule, all threads must be working on important (earliest priority) work. When a task is executed, it may create one or more new tasks with earlier priority than itself because priorities are arbitrary application-specific functions. If so, the thread executes the task with the earliest priority and adds all the other tasks to the local map. Threads search for tasks with a different priority only when the bag in which they are working becomes empty; the threads then scan the global map looking for important work. This procedure is called the *back-scan*.

Because a scan over the entire global map can be expensive, especially if there are many bags (which often happens with high-diameter graphs), an approximate consensus heuristic is used to locally estimate the earliest priority work available and to prevent redundant back-scans, which we call *back-scan prevention*. Each thread publishes the priority it is working at by writing it to shared memory. When a thread needs to scan for work, it looks at this value for all threads that share the same package and uses the earliest priority it finds to start the scan for work. To propagate information between packages, in addition to scanning all the threads in its package, one leader thread per package will scan the other package leaders. This restriction allows most threads to incur only a small amount of local communication. Once a thread has a starting point for a scan, it simply tries to pop work from each bag from the scan point onwards.

Evaluation of design choices To evaluate our design choices, we implemented several de-optimized variants of the obim scheduler. Figure 5b lists the variants, which focus on two main optimizations: (i) the use of distributed bags and (ii) back-scan prevention. We disable distributed bags by replacing the per-package lock-free stacks inside the bag with a single lock-free stack shared by all threads. We disable back-scan prevention by always starting the priority scan from the earliest priority.

Figure 6 shows the machines we used for the evaluation. The numa8x4 is an SGI Ultraviolet, which is a NUMA machine. The other machines are Intel Xeon machines with multiple packages connected by Intel's Quick Path Interconnect.

Figure 5a shows the speedup of SSSP relative to the best overall single-threaded execution time on the road graph described in Section 5.1. We can see that back-scan prevention is critical for performance: without this



(a) Speedup of variants on road graph (input details §5.1).

	Backscan Prevention	
	No	Yes
Centralized Bag	cmn	cmb
Distributed Bag	dmn	dmb (obim)

(b) Obim variants.

Figure 5: Scaling of obim and its variants for sssp.

	P	C/P	GHz	L3 (MB)	RAM (GB)	Model
m2x4	2	4	2.93	8	24	X5570
m2x8	2	8	2.70	20	32	E5-2680
m4x10	4	10	2.27	24	128	E7-4860
numa8x4	8	4	1.87	18	128	E7520

Figure 6: Evaluation machines. P is the number of packages. C/P is the number of cores per package.

optimization (cmn and dmn), speedup is never more than 2.5 on any machine for any thread count, but with this optimization (cmb and dmb), speedup rises to about 12 on 20 threads on the m4x10 machine.

Using distributed bags is also important for performance: without this optimization, speedup is never more than 5 on any machine. It is interesting to note that without back-scan prevention, a distributed bag is less efficient than a centralized one on this input. This is because it is more efficient to check that a (single) centralized bag is empty than it is to perform this check on a (per-package) distributed bag.

Related work Using concurrent priority queues for task scheduling has been explored previously [5, 13, 29, 32]. Our experience with the most common implementation, concurrent skip-lists [29], revealed poor performance on our applications. Improved performance can be achieved by using bounded priorities [30], but the basic problem of lack of scalability remains. Chazelle investigated approximate priorities [8] but only considered sequential implementations.

Another possibility is to use a concurrent priority queue for each thread, with work-stealing from other priority queues if the local priority queue becomes empty. Variations of this idea have been used previously in the literature [2, 24], and it is also used in GraphLab. However, the work efficiency of the resulting implementation is often poor because early priority work generated by one thread does not diffuse quickly enough to other threads.

Yet another possibility is to use a concurrent priority queue for each thread, with logically partitioned graphs and the owner-computes rule [27] for task assignment. This policy is well-suited for distributed systems and has been used in distributed graph traversal algorithms [25] but will perform poorly when work is localized to a subset of partitions.

4.2 Scalable library and runtime

Memory allocation Galois data structures are based on a scalable memory allocator that we implemented. While there has been considerable effort towards creating scalable memory allocators [1, 22, 28], we have found that existing solutions do not scale to large-scale multi-threaded workloads that are very allocation intensive nor do they directly address non-uniform memory access (NUMA) concerns, which are important for even modest-sized multi-core architectures.

Providing a general, scalable memory allocator is a large undertaking, particularly because Galois supports morph applications that modify graphs by adding and removing nodes and edges. For graph analytics applications, memory allocation is generally restricted to two cases: allocations in the runtime (including library data structures) and allocations in an activity to track per-activity state.

For the first case, the Galois runtime system uses a slab allocator, which allocates memory from pools of fixed-size blocks. This allocator is scalable but cannot handle variable-sized blocks efficiently due to the overhead of managing fragmentation. The second case involves allocations from user code, which may require variable-sized allocation, but also have a defined lifetime, i.e., the duration of an activity. For this case, the Galois system uses a bump-pointer region allocator.

The slab allocator has a separate allocator for each block size and a central page pool, which contains huge pages allocated from the operating system. Each thread maintains a free list of blocks. Blocks are allocated first from the free list. If the list is empty, the thread acquires a page from the page pool and uses bump-pointer allocation to divide the page into blocks.

The page pool is NUMA-aware; freed pages are returned to the region of the pool representing the memory node they were allocated from.

Allocating pages from the operating system can be a significant scalability bottleneck [9, 34], so to initialize the page pool, each application preallocates some number of pages prior to parallel execution; the exact amount varies by application.

The bump-pointer allocator manages allocations of per-activity data structures, which come from temporaries created by user code in the operator. This allocator supports standard C++ allocator semantics, making it usable with all standard containers. The allocator is backed by a page from the page pool. If the allocation size exceeds the page size (2 MB), the allocator falls back to malloc.

Each activity executes on a thread, which has its own instance of the bump-pointer allocator. The allocator is reused (after being reset) between iterations on a thread. Since the lifetimes of the objects allocated are bound to an activity, all memory can be reclaimed at once at the end of the activity.

Topology-aware synchronization The runtime library performs optimizations to avoid synchronization and communication between threads. Communication patterns are topology-aware, so that the most common synchronization is only between cores on the same package and share the same L3 cache. Communication between these cores is cheap. For example, instead of using a standard Pthread barrier, the runtime uses a hybrid barrier where a tree topology is built across packages, but threads in a package communicate via a shared counter. This is about twice as fast as the classic MCS tree barrier [20]. A similar technique, signaling trees, can be applied to signal threads to execute, which is used to begin parallel execution.

Code size optimizations Galois provides a rich programming model that requires memory allocation, task scheduling, application-specific task priorities, and speculative execution. Each of these features incurs some runtime cost. In general, since tasks can create new tasks, the support code for an operator must check if new tasks were created and if so hand them to the scheduler. However, the operators for many algorithms do not create new tasks; although this check requires only about 4 instructions (at least one of which is a load and one of which is a branch), this amounts to almost 2% of the number of instructions in an average SSSP task. For applications with fine-grain tasks, generality can quickly choke performance.

To reduce this overhead, Galois does not generate code for features that are not used by an operator. It uses a collection of type traits that statically communicate the

Feature	LoC
Vertex Programs	0
Gather-Apply-Scatter (synchronous engine)	200
Gather-Apply-Scatter (asynchronous engine)	200
Out-of-core	400
Push-versus-pull	300
Out-of-core + Push-versus-pull (additional)	100

Figure 7: Approximate lines of code for each DSL feature.

features of the programming model not needed by an operator to the runtime system. At compile time, a specialized implementation for each operator is generated that only supports the required features.

This simplification of runtime code has an important secondary effect on performance: tight loops are more likely to fit in the trace cache or the L1 instruction cache. For very short operators, such as the one in BFS or SSSP, this can result in a sizable performance improvement.

These kinds of code specialization optimizations reduce the overhead of dynamic task creation, priority scheduling and load balancing for SSSP to about 140 instructions per activity; about half of which comes from the priority scheduler.

4.3 Other DSLs in Galois

By using the features described above, graph analytics DSLs such GraphLab, GraphChi and Ligra can be simply layered on top of Galois. Also, to demonstrate the ease with which new DSLs can be implemented, we also implemented a DSL that combines features of the Ligra and GraphChi systems. Figure 7 gives the approximate lines of code required to implement these features on top of the Galois system.

Vertex Programs These are directly supported by Galois. Granularity of serializability can be controlled through the use of Galois data structure parameters (§4). For example, to achieve the GraphLab edge consistency model, a user can enable logical locks when accessing a vertex and its edges but not acquire logical locks when accessing a neighboring node.

Gather-apply-scatter The PowerGraph implementation of GAS programs has three different execution models: coordinated scheduling, autonomous scheduling without consistency guarantees, and autonomous scheduling with serializable activities. We call the Galois implementation of PowerGraph **PowerGraph-g**. The two autonomous scheduling models can be implemented in Galois by concatenating the gather, apply and scatter steps for a vertex into a single Galois operator, and either always or never acquiring logical locks during the execution of the operator.

The coordinated scheduling model can be implemented by a sequence of loops, one for each phase of the GAS programming model. The main implementation question is how to implement the scatter phase, since we must handle the case when multiple nodes send messages to the same neighbor. The implementation we settled on is to have per-package message accumulation, protected by a spin-lock. The receiver accumulates the final message value by reading from the per-package locations. PowerGraph supports dividing up the work of a single gather or scatter operation for a node among different processing units. PowerGraph-g does not yet have this optimization.

Out-of-core The GraphChi implementation of out-of-core processing for vertex programs uses a carefully designed graph file format to support IO-efficient access to both the incoming and outgoing edge values of a node. For the purpose of understanding how to provide out-of-core processing using general reusable components, we focus on supporting only a subset of GraphChi features.

Our implementation of out-of-core processing is based on incremental loading of the compressed sparse row (CSR) format of a graph and the graph's transpose. The transpose graph represents the incoming edges of a graph and stores a copy of the edge values of the corresponding outgoing edges. Since these values are copies, we do not support updating edge values like GraphChi does; however, none of the applications described in this paper require updating edge values, and we leave supporting this feature to future work.

Push-versus-pull The push-versus-pull optimization in Ligra can be implemented as two vertex programs that take an edge update rule and perform either a forward or backward traversal based on some threshold. We call the Galois implementation of Ligra **Ligra-g**. We use the same threshold heuristic as Ligra. In order to perform this optimization, the graph representation must store both incoming and outgoing edges.

Push-versus-pull and out-of-core Since the push-versus-pull optimization is itself a vertex program, we can compose the push-versus-pull vertex program with the out-of-core processing described above. We call the DSL that combines both these optimizations **LigraChi-g**. This highlights the utility of having a single framework for implementing DSLs.

	$\|V\|$	$\|E\|$	MB	Weighted MB
rmat24	17	268	1207	2281
rmat27	134	2141	9637	18218
twitter40	42	1469	6207	12080
twitter50	51	1963	8262	16114
road	24	58	422	653

Figure 8: Input characteristics. Number of nodes and edges is in millions. MB is the size of CSR representation.

5 Evaluation

In this section, we compare the performance of applications (§3) in the Ligra, GraphLab v1, PowerGraph v2.1 and Galois v2.2 systems.

5.1 Methodology

Our evaluation machine is machine m4x10 (see Figure 6). PowerGraph is a distributed-memory implementation, but it supports shared-memory parallelism within a single machine. Ligra, GraphLab and Galois are strictly shared-memory systems. Ligra requires the Cilk runtime, which is not yet available with the GCC compiler. We compiled Ligra with the Intel ICC 12.1 compiler. All other applications were compiled with GCC 4.7 with the -O3 optimization level.

All runtimes are an average of at least two runs. For out-of-core DSLs, the runtimes include the time to load data from local disk. For all other systems, we exclude this time.

Figure 8 summarizes the graph inputs we used. The rmat24 ($a = 0.5$, $b = c = 0.1$, $d = 0.3$) and rmat27 ($a = 0.57$, $b = c = 0.19$, $d = 0.05$) graphs are synthetic scale-free graphs. Following the evaluation done by Shun and Blelloch [31], we made the graphs symmetric. The twitter40 [16] and twitter50 [6] graphs are real-world social network graphs. From twitter40 and twitter50, we use only the largest connected component. Finally, road is a road network of the United States obtained from the DIMACS shortest paths benchmark.

The road graph is naturally weighted; we simply remove the weights when we need an unweighted graph. The other four graphs are unweighted. To provide a weighted input for the SSSP algorithms, we add a random edge weight in the range $(0, 100]$ to each edge.

5.2 Summary of results

Figure 9a shows the runtime ratios of the Ligra and PowerGraph applications compared to the Galois versions on the twitter50 and road inputs for five applications. When the Galois version runs faster, the data point is shown as a cross; otherwise it is shown as a circle (e.g., bfs on twitter50 and pr on road). The values range over several orders of magnitude. The largest improvements are on the road graph and with respect to PowerGraph.

Figure 9b shows the runtime ratios of the Ligra and PowerGraph applications compared to the Ligra-g and PowerGraph-g versions (that is, the implementations of those DSLs in Galois). The performance of Ligra-g is roughly comparable to Ligra. PowerGraph-g is mostly better than PowerGraph. This shows that much of the huge improvements in Figure 9a come not so much from the better implementations described in Section 4 *per se* but from the better programs that can be written when the programming model is rich enough.

Comparison between the two figures can also be illuminating. For example, most of the ratios in Figure 9a are greater than those in Figure 9b, but one notable exception is the behavior of PowerGraph with PageRank on the road graph. The Galois improvement is about 10X while the PowerGraph-g improvement is about 50X. This suggests that the Galois application of PageRank, which is pull-based, is not as good as the push-based algorithm used by PowerGraph, on the road graph. Thus, Galois is faster than PowerGraph on PageRank because of a more efficient implementation of a worse algorithm.

In the following sections, we dig deeper into our performance results.

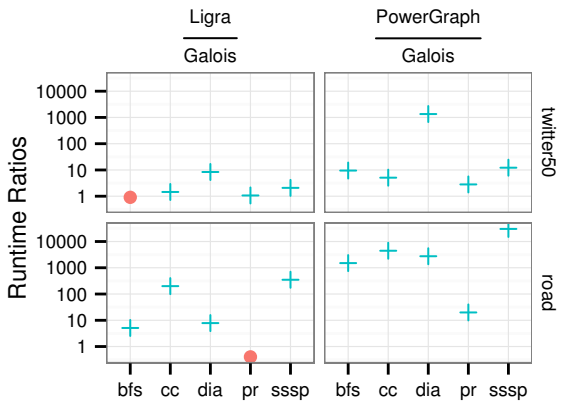
5.3 Complete results

Figure 10 gives our complete runtime results with 40 threads. The Ligra-g and PowerGraph-g results will be discussed in Section 5.4. The PageRank (pr) times are for one iteration of the topology-driven algorithm.

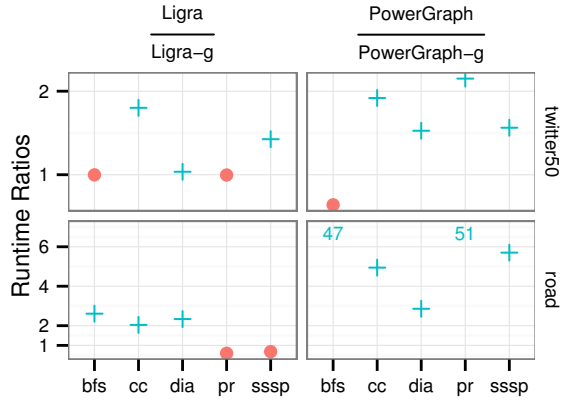
Overall, there is a wide variation in running times across different programming models solving the same problem. The variation is the least for PageRank, which is coordinated and topology-driven. For the other graph problems, the performance difference between programming models can be several orders of magnitude and are quite stark for the road input, whose large diameter heavily penalizes coordinated scheduling of data-driven algorithms.

The performance differences can be broadly attributed to three causes.

In some cases, a poor algorithm was selected even though a better algorithm exists and is expressible in the DSL. The PowerGraph diameter application is an example of this. The probabilistic counting algorithm just takes too long on these inputs and gives worse results than the Ligra algorithm, which also can be expressed as a gather-apply-scatter program. Figure 11 gives the approximate diameters returned by each algorithm. The PowerGraph algorithm quits after trying diameters up to



(a) Ratio of Ligra and PowerGraph runtimes to Galois runtimes.



(b) Ratio of Ligra and PowerGraph runtimes relative to Ligra-g and PowerGraph-g runtimes. Larger ratios shown as numbers rather than points.

Figure 9: Ratio of runtimes of applications with 40 threads on machine m4x10 (see Figure 10 for runtimes). Values greater than 1 (shown as blue crosses) indicate how many times faster the denominator system is than the numerator.

100. Both Galois and Ligra algorithms give strict lower-bounds on the true diameter. The PowerGraph algorithm gives a probabilistic estimate.

In some other cases, the same algorithm is expressed in multiple DSLs, but one programming model just has a better system implementation. All the PageRank applications are largely implementations of the same algorithm. Differences between implementations are due to differences in the runtime systems for each programming model.

Finally, a DSL may be unable to capture an important algorithmic optimization—such as when an important optimization simply cannot be expressed in a DSL or when the optimization can be expressed but the implementation of the DSL cannot adequately exploit it.

		rmat24	rmat27	twitter40	twitter50	road
bfs	Galois	0.5	1.5	**0.7**	2.5	**0.5**
bfs	Ligra-g	**0.3**	**1.3**	0.8	2.3	1.1
bfs	PowerGraph-g	10.8	84.2	28.0	37.7	17.5
bfs	GraphLab	12.4	83.9	26.7	60.5	4092.7
bfs	Ligra	0.4	1.5	1.2	**2.3**	2.8
bfs	PowerGraph	7.0	30.8	16.9	24.1	821.6
cc	Galois	7.3	17.9	13.9†	39.6†	**0.6**
cc	Ligra-g	**1.3**	**11.1**	16.6	31.9	62.3
cc	PowerGraph-g	21.8	120.3	58.8	105.0	572.9
cc	GraphLab	14.1	89.6	36.0	64.5	1033.5
cc	Ligra	2.5	22.2	31.7	57.5	127.0
cc	PowerGraph	39.0	129.5	115.5	201.5	2831.5
dia	Galois	**1.1**	**5.1**	**2.8**	**5.5**	**2.6**
dia	Ligra-g	2.3	21.4	19.7	44.3	8.6
dia	PowerGraph-g	2029.7	oom	3816.1	4841.9	2466.6
dia	GraphLab	84.8	478.2	192.0	257.1	21363.3
dia	Ligra	1.7	11.8	19.3	45.8	20.1
dia	PowerGraph	1239.0	oom	5376.0	7390.5	7047.5
pr	Galois	1.3	**10.3**	6.5	**10.7**	0.5
pr	Ligra-g	1.1	15.6	**4.6**	11.5	0.4
pr	PowerGraph-g	2.1	21.0	11.7	14.0	**0.2**
pr	GraphLab	4.9	47.6	45.8	30.7	14.6
pr	Ligra	**1.0**	11.6	8.7	11.5	0.2
pr	PowerGraph	8.4	38.8	20.4	30.2	10.6
sssp	Galois	**1.9**	**6.0**	11.6	**8.6**	**0.6**
sssp	Ligra-g	2.8	9.1	**10.0**	12.5	320.7
sssp	PowerGraph-g	22.8	100.0	43.3	66.8	3317.3
sssp	GraphLab	28.8	153.9	60.9	87.6	28.6
sssp	Ligra	2.3	12.3	15.9	17.8	219.4
sssp	PowerGraph	34.4	78.8	52.9	104.4	18919.2
bc	Galois	1.3	13.7	13.0	**12.0**	**1.3**
bc	Ligra-g	1.4	7.6	**5.3**	12.9	5.1
bc	Ligra	**1.2**	**5.5**	6.8	13.9	6.6

Figure 10: Runtime in seconds of applications with 40 threads on machine m4x10. The label oom indicates the application ran out of memory. In bold are the best times for each input and graph problem pair. (†) indicates that the best time on cc occurred with eight threads: twitter40 (13.8 s), twitter50 (13.6 s).

	rmat24	rmat27	twitter40	twitter50	road
Galois	17	10	14	14	8440
Ligra	10	6	15	15	6262
PowerGraph	9		7	8	>100

Figure 11: Approximate diameters computed.

	rmat24		rmat27		twitter40		twitter50		road	
	8x16	64x16	8x16	64x16	8x16	64x16	8x16	64x16	8x16	64x16
bfs	29.2	21.8	73.0	28.6	73.2	38.5	81.5	50.8	161.5	821.6
cc	114.5	53.5	270.5	71.5	270.0	90.0	406.0	112.0		
pr	11.6	9.9	43.2	14.8	30.5	17.6	42.3	16.4	4.1	5.8
sssp	112.0	76.9	173.9	63.0	175.2	111.0	321.9	127.5		

Figure 12: Runtime in seconds of PowerGraph applications on a distributed system with eight or 64 m2x8 machines.

An example of not being able to express an optimization is the lack of priority scheduling for the Ligra and PowerGraph applications for SSSP. GraphLab supports priority scheduling, so although the GraphLab SSSP application is worse on scale-free inputs, it performs much better than Ligra and PowerGraph on the road input due to its support for priority scheduling. Thus, in some cases, it is preferable to have inefficient support for priority scheduling than no support at all.

Another example is the push-versus-pull optimization implemented in Ligra. In principle, this optimization can be implemented in any DSL that supports coordinated scheduling of vertex programs, like GraphLab, but GraphLab does not provide any support for user-visible concurrent bag or worklist objects, so it is not possible to efficiently switch between push and pull traversals.

An example of the inability to exploit an optimization is the GraphLab diameter application. We implemented the faster pseudo-peripheral algorithm in GraphLab, but because of large overheads starting and stopping parallel execution, which are required for the sequential composition of the parallel breadth-first searches, the overall application has very poor performance.

Figure 12 shows the performance of PowerGraph when run on a distributed system, the Stampede cluster at the Texas Advanced Computing Center (TACC). Each machine that we used in the cluster is an instance of machine m2x8 in Figure 6. Given the poor performance of the connected components and SSSP PowerGraph implementations on the road graph on shared-memory machines, we elected not to run those applications on the distributed machine. Even with 64 machines ($64 \cdot 16 = 1024$ cores), the performance is worse than that of the best implementation on a single machine with 8 cores and 4 times the RAM for all but one application-input combination (data not shown here). The one slower combination is PageRank on rmat27 where Galois takes 15.6 seconds and PowerGraph takes 14.8 seconds.

5.4 Comparison of implementations

Figure 10 also shows the performance results of Galois versions of Ligra and PowerGraph—Ligra-g and PowerGraph-g, respectively.

Overall, the Galois implementations of graph DSLs do better than the original DSL implementations, although this varies from DSL to DSL. If we consider pairs of Ligra and Ligra-g runtimes for each graph problem, input and number of threads, in $18/30 \approx 60\%$ of the pairs, the Galois version is faster. Note that, due to incompatibilities, a different compiler was used for each version of the application. Considering pairs of PowerGraph and PowerGraph-g, runtimes, $18/24 = 0.75\%$ of the pairs favor the Galois version. As noted earlier, PowerGraph supports distributed-memory execution as well, so some portion of the performance gap is due to the additional overhead of supporting distributed-memory execution and not using it. For GraphLab and Galois, all of the pairs favor Galois, although in this case, the Galois applications include optimizations that could not be implemented in GraphLab, like push-versus-pull.

Some improvements can be made to the Galois versions of these DSLs. For instance, the Ligra version of the diameter application tends to be faster than the Ligra-g version, and the PowerGraph version of BFS tends to scale better than the PowerGraph-g version, but overall, the results suggest that the Galois infrastructure is a reasonable substrate on which graph DSLs can be built.

5.5 Evaluation of out-of-core DSLs

To evaluate the out-of-core DSLs, GraphChi and our combination of Ligra and GraphChi that we call **LigraChi-g**, we use a machine with less memory, m2x4 (see Figure 6). To test the out-of-core capability, we give each system a memory budget of 2 GB of RAM to store graph data. This includes graph adjacency information and edge values, but it does not include user data allocated for a node nor any additional user or runtime-allocated structures. The entire road graph fits in this memory budget.

		rmat24	rmat27	twitter40	twitter50	road
bfs	LigraChi-g	9	133	187	960	12
cc	LigraChi-g	17	205	175	310	169
cc	GraphChi	223	1164	870	1179	120
dia	LigraChi-g	21	192	265	697	29
pr	LigraChi-g	16	143	90	114	6
pr	GraphChi	38	308	154	220	13
sssp	LigraChi-g	36	1127	3227	4873	790
bc	LigraChi-g	18	237	251	1561	14

Figure 13: Runtime in seconds of applications on machine m2x4 with eight threads.

Figure 13 gives the performance of the out-of-core DSLs, GraphChi and LigraChi-g. Inputs were stored on a 7200 RPM SATA drive. GraphChi allows separate configuration of load threads, which read the graph file, and execute threads, which run the vertex program. For these experiments, the number of threads refers to the number of execute threads. We always use two load threads.

These out-of-core experiments highlight the impact of having enough memory for graph analytics applications. Ignoring differences in processors but keeping the number of threads the same, on the larger inputs, i.e., rmat27, twitter40 and twitter50, running in a memory-constrained environment with LigraChi-g (see Figure 13) is between 3.4X and 197X slower than performing the same algorithm with Ligra-g on machine m4x10 (data not shown here), an unconstrained memory environment.

These results provide more context for the recent claim by Kyrola et al. that out-of-core execution of graph analytics only incurs a modest performance penalty [17].

The slowdown is between 3.4X and 7.8X for the connected components, approximate diameter and PageRank applications. With a 5X reduction in memory, these results suggest a reasonable trade-off between space and time for connected components and PageRank. For the approximate diameter application, there are additional gains from switching to the more expressive Galois programming model.

For the other applications, the slowdown ranges between 11.5X and 197X, not including the additional slowdown of Ligra or Ligra-g versus Galois. The out-of-core DSLs impose a particular scheduling of activities that optimizes IO operations, but that order may not be efficient from the application standpoint. For more effective out-of-core implementations of these applications, more attention should be paid towards the joint optimization of application and IO scheduling.

6 Conclusion

A number of DSLs for graph analytics have been proposed recently. Given the importance of the problem domain and the limitations of existing DSLs, it is likely that more DSLs will be designed and implemented in the near future. In this paper, we argued that these DSLs require a lightweight infrastructure that supports autonomous scheduling of fine-grain tasks with application-specific priorities. We presented the design and implementation of the Galois system, which accomplishes this through a machine-topology-aware scheduler, a priority scheduler called obim, and a library of scalable data structures.

We demonstrated the capabilities of the Galois infrastructure in three ways. First, we implemented sophisticated algorithms for some of the graph analytics problems, argued that they cannot be implemented in existing DSLs, and showed that end-to-end performance is improved by many orders of magnitude, thanks to the better algorithms. Second, we showed that even when an algorithm can be expressed in existing DSLs, its implementation in Galois can be orders of magnitude faster when the input graphs are road networks and similar graphs with high diameter, thanks to better scheduling. Third, we implemented the APIs of three existing graph DSLs on top of the common infrastructure in a few hundred lines of code and showed that even for power-law graphs, the performance of the resulting implementations often exceeds that of the original DSL systems, thanks to the lightweight infrastructure. Furthermore, we also showed that combinations of features from some of these DSLs can be implemented easily on top of the Galois system.

From our experience with graph analytics programs, we offer the following lessons.

To those interested in graph analytics: Far greater performance gains, often by orders of magnitude, can come from choosing an expressive programming model than can be obtained by tuning a restricted DSL. In this domain, a rich programming model with an inefficient implementation can often outperform a more restricted programming model with a more efficient implementation!

To implementers of DSLs on shared-memory systems: Exploiting machine topology and scalable memory allocation are key to scalable implementations.

To designers of programming models on distributed-memory systems: Just as in shared-memory systems, the large improvements possible from the selection of a rich programming model should be an important consideration for the design and implementation of graph analytics on distributed memory.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, Nov. 2000.

[2] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *J. Optim. Theory Appl.*, 88(2):297–320, Feb. 1996.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[4] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

[5] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. ACM SIGPLAN Symp. Principles And Practice of Parallel Programming*, PPoPP '10, pages 257–268, 2010.

[6] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. Intl AAAI Conf. Weblogs and Social Media*, ICWSM '10, 2010.

[7] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, Oct. 1984.

[8] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, Nov. 2000.

[9] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 199–210, 2012.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proc. USENIX Conf. Operating Systems Design and Implementation*, OSDI '12, pages 17–30, 2012.

[12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, 2012.

[13] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60:151–157, November 1996.

[14] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '07, pages 211–222, 2007.

[16] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. Intl Conf. World Wide Web*, WWW '10, pages 591–600, 2010.

[17] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proc. USENIX Conf. Operating Systems Design and Implementation*, OSDI '12, pages 31–46, 2012.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proc. Conf. Uncertainty in Artificial Intelligence*, UAI '10, July 2010.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Intl Conf. on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[20] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[21] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *Proc. European Symposium on Algorithms*, ESA '98, pages 393–404, 1998.

[22] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '04, pages 35–46, 2004.

[23] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 333–344, 2011.

[24] M. Papaefthymiou and J. Rodrigue. Implementing parallel shortest-paths algorithms. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–68, 1994.

[25] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proc. ACM/IEEE Intl Conf. High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[26] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[27] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '89, pages 69–80, 1989.

[28] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. Intl Symp. Memory Management*, ISMM '06, pages 84–94, 2006.

[29] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. Intl Parallel and Distributed Processing Symp./Intl Parallel Processing Symp.*, IPDPS '00, pages 263–268, 2000.

[30] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proc. ACM Symp. Principles of Distributed Computing*, PODC '99, pages 113–122, 1999.

[31] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[32] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65:609–627, May 2005.

[33] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[34] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. IEEE Intl Symp. Workload Characterization*, IISWC '09, pages 198–207, 2009.