Pregelix: Big(ger) Graph Analytics on A Dataflow Engine

Yingyi Bu[1] Vinayak Borkar[2*] Jianfeng Jia[1] Michael J. Carey[1] Tyson Condie[3]
[1]University of California, Irvine [2]X15 Software, Inc [3]University of California, Los Angeles
[1]yingyib,jianfenj,mjcarey@ics.uci.edu, [2]vinayakb@x15soft.com, [3]tcondie@cs.ucla.edu

ABSTRACT

There is a growing need for distributed graph processing systems that are capable of gracefully scaling to very large graph datasets. Unfortunately, this challenge has not been easily met due to the intense memory pressure imposed by process-centric, message passing designs that many graph processing systems follow. Pregelix is a new open source distributed graph processing system that is based on an iterative dataflow design that is better tuned to handle both in-memory and out-of-core workloads. As such, Pregelix offers improved performance characteristics and scaling properties over current open source systems (e.g., we have seen up to 15× speedup compared to Apache Giraph and up to 35× speedup compared to distributed GraphLab), and more effective use of available machine resources to support Big(ger) Graph Analytics.

1. INTRODUCTION

There are increasing demands to process Big Graphs for applications in social networking (e.g., friend recommendations), the web (e.g., ranking pages), and human genome assembly (e.g., extracting gene sequences). Unfortunately, the basic toolkits provided by first-generation "Big Data" Analytics platforms (like Hadoop) lack an essential feature for Big Graph Analytics: MapReduce does not support iteration (or equivalently, recursion) or certain key features required to efficiently iterate "around" a MapReduce program. Moreover, the MapReduce programming model is not ideal for expressing many graph algorithms. This shortcoming has motivated several specialized approaches or libraries that provide support for graph-based iterative programming on large clusters.

Google's Pregel is a prototypical example of such a platform; it allows problem-solvers to "think like a vertex" by writing a few user-defined functions (UDFs) that operate on vertices, which the framework can then apply to an arbitrarily large graph in a parallel fashion. Open source versions of Pregel have since been developed in the systems community [4, 6]. Perhaps unfortunately for both their implementors and users, each such platform is a distinct new system that had to be built from the ground up. Moreover, these systems follow a process-centric design, in which a set of worker processes are assigned partitions (containing sub-graphs) of the graph data and scheduled across a machine cluster. When a worker process launches, it reads its assigned partition into memory, and executes the Pregel (message passing) algorithm. As we will see, such a design can suffer from poor support for problems that are not memory resident. Also desirable, would be the ability to consider alternative runtime strategies that could offer more efficient executions for different sorts of graph algorithms, datasets, and clusters.

The database community has spent nearly three decades building efficient shared-nothing parallel query execution engines [22] that support out-of-core data processing operators (such as join and group-by [27]), and query optimizers [16] that choose an "optimal" execution plan among different alternatives. In addition, deductive database systems—based on Datalog—were proposed to efficiently process recursive queries [10], which can be used to solve graph problems such as transitive closure. However, there is no scalable implementation of Datalog that offers the same fault-tolerant properties supported by today's "Big Data" systems (e.g., [32, 5, 11, 12, 44]). Nevertheless, techniques for evaluating recursive queries—most notably semi-naïve evaluation—still apply and can be used to implement a scalable, fault-tolerant Pregel runtime.

In this paper, we present Pregelix, a large-scale graph analytics system that we began building in 2011. Pregelix takes a novel set-oriented, iterative dataflow approach to implementing the user-level Pregel programming model. It does so by treating the messages and vertex states in a Pregel computation like tuples with a well-defined schema; it then uses database-style query evaluation techniques to execute the user's program. From a user's perspective, Pregelix provides the same Pregel programming abstraction, just like Giraph [4]. However, from a runtime perspective, Pregelix models Pregel's semantics as a logical query plan and implements those semantics as an iterative dataflow of relational operators that treat message exchange as a join followed by a group-by operation that embeds functions that capture the user's Pregel program. By taking this approach, for the same logical plan, Pregelix is able to offer a set of alternative physical evaluation strategies that can fit various workloads and can be executed by Hyracks [12], a general-purpose shared-nothing dataflow engine (which is also the query execution engine for AsterixDB [1]). By leveraging existing implementations of data-parallel operators and access methods from Hyracks, we have avoided building many critical system components, e.g., bulk-data network transfer, out-of-core operator implementations, buffer managers, index structures, and data shuffle.

To the best of our knowledge, Pregelix is the only Pregel-like system that supports the full Pregel API, runs both in-memory workloads and out-of-core workloads efficiently in a transparent

*work done at the University of California, Irvine.

Proceedings of the VLDB Endowment, Vol. 8, No. 2

manner on shared-nothing clusters, and provides a rich set of runtime choices. This paper makes the following contributions:

- An analysis of existing Pregel-like systems: We revisit the Pregel programming abstraction and illustrate some shortcomings of typical custom-constructed Pregel-like systems (Section 2).
- A new Pregel architecture: We capture the semantics of Pregel in a logical query plan (Section 3), allowing us to execute Pregel as an iterative dataflow.
- A system implementation: We first review the relevant building blocks in Hyracks (Section 4). We then present the Pregelix system, elaborating the choices of data storage and physical plans as well as its key implementation details (Section 5).
- Case studies: We briefly describe several current use cases of Pregelix from our initial user community (Section 6).
- Experimental studies: We experimentally evaluate Pregelix in terms of execution time, scalability, throughput, plan flexibility, and implementation effort (Section 7).

2. BACKGROUND AND PROBLEMS

In this section, we first briefly revisit the Pregel semantics and the Google Pregel runtime (Section 2.1) as well as the internals of Giraph, an open source Pregel-like system (Section 2.2), and then discuss the shortcomings of such custom-constructed Pregel architectures (Section 2.3).

2.1 Pregel Semantics and Runtime

Pregel [32] was inspired by Valiant's bulk-synchronous parallel (BSP) model [26]. A Pregel program describes a distributed graph algorithm in terms of vertices, edges, and a sequence of iterations called supersteps. The input to a Pregel computation is a directed graph consisting of edges and vertices; each vertex is associated with a mutable user-defined value and a boolean state indicating its liveness; each edge is associated with a source and destination vertex and a mutable user-defined value. During a superstep S, a user-defined function (UDF) called `compute` is executed at each active vertex V, and can perform any or all of the following actions:

- Read the messages sent to V at the end of superstep $S-1$;
- Generate messages for other vertices, which will be exchanged at the end of superstep S;
- Modify the state of V and its outgoing edges;
- Mutate the graph topology;
- Deactivate V from the execution.

Initially, all vertices are in the active state. A vertex can deactivate itself by "voting to halt" in the call to `compute` using a Pregel provided method. A vertex is reactivated immediately if it receives a message. A Pregel program terminates when every vertex is in the inactive state and no messages are in flight.

In a given superstep, any number of messages may be sent to a given destination. A user-defined `combine` function can be used to pre-aggregate the messages for a destination. In addition, an aggregation function (e.g., min, max, sum, etc.) can be used to compute a global aggregate among a set of participating vertices. Finally, the graph structure can be modified by any vertex; conflicts are handled by using a partial ordering of operations such that all deletions go before insertions, and then by using a user-defined conflict resolution function.

The Google Pregel runtime consists of a centralized master node that coordinates superstep executions on a cluster of worker nodes. At the beginning of a Pregel job, each worker loads an assigned graph partition from a distributed file system. During execution, each worker calls the user-defined `compute` function on each active vertex in its partition, passing in any messages sent to the vertex in the previous superstep; outgoing messages are exchanged among workers. The master is responsible for coordinating supersteps and detecting termination. Fault-tolerance is achieved through checkpointing at user-specified superstep boundaries.

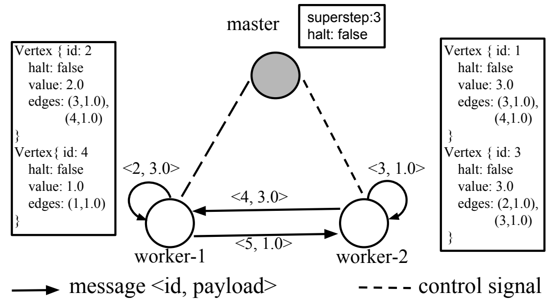


Figure 1: Giraph process-centric runtime.

2.2 Apache Giraph

Apache Giraph [4] is an open source project that implements the Pregel specification in Java on the Hadoop infrastructure. Giraph launches master and worker instances in a Hadoop map-only job[1], where map tasks run master and worker instances. Once started, the master and worker map tasks internally execute the iterative computation until completion, in a similar manner to Google's Pregel runtime. Figure 1 depicts Giraph's process-centric runtime for implementing the Pregel programming model. The vertex data is partitioned across worker tasks (two in this case). Each worker task communicates its control state (e.g., how many active vertices it owns, when it has completed executing a given superstep, etc.) to the master task. The worker tasks establish communication channels between one another for exchanging messages that get sent during individual vertex `compute` calls; some of these messages could be for vertices on the same worker, e.g., messages <2, 3.0> and <3,1.0> in Figure 1.

2.3 Issues and Opportunities

Most process-centric Pregel-like systems have a minimum requirement for the aggregate RAM needed to run a given algorithm on a particular dataset, making them hard to configure for memory intensive computations and multi-user workloads. In fact, Google's Pregel only supports in-memory computations, as stated in the original paper [32]. Hama [6] has limited support for out-of-core vertex storage using immutable sorted files, but it requires that the messages be memory-resident. The latest version of Giraph has preliminary out-of-core support; however, as we will see in Section 7, it does not yet work as expected. Moreover, in the Giraph user mailing list[2] there are 26 cases (among 350 in total) of out-of-memory related issues from March 2013 to March 2014. The users who posted those questions were typically from academic institutes or small businesses that could not afford memory-rich clusters, but who still wanted to analyze Big Graphs. These issues essentially stem from Giraph's ad-hoc, custom-constructed implementation of disk-based graph processing. This leads to our first opportunity to improve on the current state-of-the-art.

Opportunity (Out-of-core Support) Can we leverage mature database-style storage management and query evaluation techniques to provide better support for out-of-core workloads?

Another aspect of process-centric designs is that they only offer a single physical layer implementation. In those systems, the vertex

[1] Alternatively, Giraph can use YARN [39] for resource allocation.
[2] http://mail-archives.apache.org/mod_mbox/giraph-user/

Relation	Schema
Vertex	(vid, halt, value, edges)
Msg	(vid, payload)
GS	(halt, aggregate, superstep)

Table 1: Nested relational schema that models the Pregel state.

storage strategy, the message combination algorithm, the message redistribution strategy, and the message delivery mechanism are each usually bound to one specific implementation. Therefore, we cannot choose between alternative implementation strategies that would offer a better fit to a particular dataset, algorithm, cluster or desktop. For instance, the single source shortest paths algorithm exhibits sparsity of messages, in which case a desired runtime strategy could avoid iterating over all vertices by using an extra index to keep track of live vertices. This leads to our second opportunity.

Opportunity (Physical Flexibility) Can we better leverage data, algorithmic, and cluster/hardware properties to optimize a specific Pregel program?

The third issue is that the implementation of a process-centric runtime for the Pregel model spans a full stack of network management, communication protocol, vertex storage, message delivery and combination, memory management, and fault-tolerance; the result is a complex (and hard-to-get-right) runtime system that implements an elegantly simple Pregel semantics. This leads to our third, software engineering opportunity.

Opportunity (Software Simplicity) Can we leverage more from existing data-parallel platforms—platforms that have been improved for many years—to simplify the implementation of a Pregel-like system?

We will see how these opportunities are exploited by our proposed architecture and implementation in Section 5.8.

3. THE PREGEL LOGICAL PLAN

In this section, we model the semantics of Pregel as a logical query plan. This model will guide the detailed design of the Pregelix system (Section 5).

Our high level approach is to treat messages and vertices as data tuples and use a join operation to model the message passing between vertices, as depicted in Figure 2. Table 1 defines a set of nested relations that we use to model the state of a Pregel execution. The input data is modeled as an instance of the `Vertex` relation; each row identifies a single vertex with its halt, value, and edge states. All vertices with a $halt = false$ state are active in the current superstep. The value and edges attributes represent the vertex state and neighbor list, which can each be of a user-defined type. The messages exchanged between vertices in a superstep are modeled by an instance of the `Msg` relation, which associates a destination vertex identifier with a message payload. Finally, the `GS` relation from Table 1 models the global state of the Pregel program; here, when $halt = true$ the program terminates[3], *aggregate* is a global state value, and *superstep* tracks the current iteration count.

Figure 2 models message passing as a join between the `Msg` and `Vertex` relations. A full-outer-join is used to match messages with vertices corresponding to the Pregel semantics as follows:

- The inner case matches incoming messages with existing destination vertices;
- The left-outer case captures messages sent to vertices that may not exist; in this case, a vertex with the given *vid* is created with other fields set to NULL.

[3]This global halting state depends on the halting states of all vertices as well as the existence of messages.

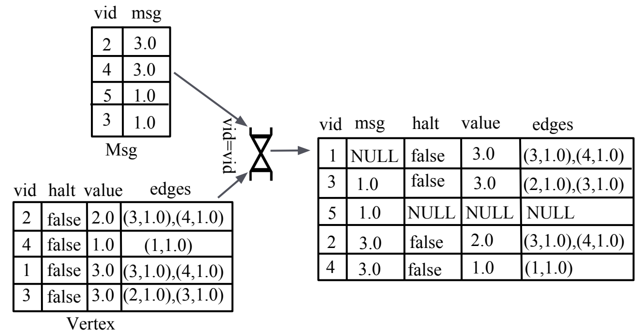


Figure 2: Implementing message-passing as a logical join.

UDF	Description
compute	Executed at each active vertex in every superstep.
combine	Aggregation function for messages.
aggregate	Aggregation function for the global state.
resolve	Used to resolve conflicts in graph mutations.

Table 2: UDFs used to capture a Pregel program.

- The right-outer case captures vertices that have no messages; in this case, `compute` still needs to be called for such a vertex if it is active.

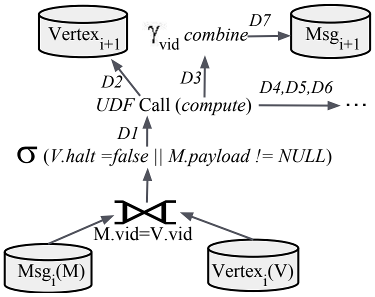
The output of the full-outer-join will be sent to further operator processing steps that implement the Pregel semantics; some of these downstream operators will involve UDFs that capture the details (e.g., `compute` implementation) of the given Pregel program.

Table 2 lists the UDFs that implement a given Pregel program. In a given superstep, each active vertex is processed through a call to the `compute` UDF, which is passed the messages sent to the vertex in the previous superstep. The output of a call to `compute` is a tuple that contains the following fields:

- The possibly updated `Vertex` tuple.
- A list of outbound messages (delivered in the next superstep).
- The global *halt* state contribution, which is *true* when the outbound message list is empty and the *halt* field of the updated vertex is *true*, and *false* otherwise.
- The global *aggregate* state contribution (tuples nested in bag).
- The graph mutations (a nested bag of tuples to insert/delete to/from the `Vertex` relation).

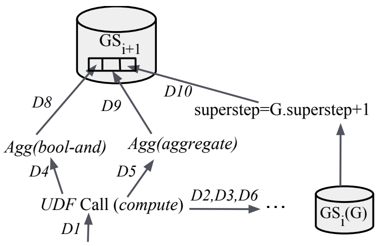
As we will see below, this output is routed to downstream operators that extract (project) one or more of these fields and execute the dataflow of a superstep. For instance, output messages are grouped by the destination vertex id and aggregated by the `combine` UDF. The global aggregate state contributions of all vertices are passed to the `aggregate` function, which produces the global aggregate state value for the subsequent superstep. Finally, the `resolve` UDF accepts all graph mutations—expressed as insertion/deletion tuples against the `Vertex` relation—as input, and it resolves any conflicts before they are applied to the `Vertex` relation.

We now turn to the description of a single logical dataflow plan; we divide it into three figures that each focus on a specific application of the (shared) output of the `compute` function. The relevant dataflows are labeled in each figure. Figure 3 defines the input to the `compute` UDF, the output messages, and updated vertices. Flow $D1$ describes the `compute` input for superstep i as being the output of a full-outer-join between `Msg` and `Vertex` (as described by Figure 2) followed by a selection predicate that prunes inactive vertices. The `compute` output pertaining to vertex data is projected onto dataflow $D2$, which then updates the `Vertex` relation.



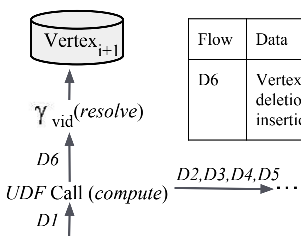
Flow	Data
D2	Vertex tuples
D3	Msg tuples
D7	Msg tuples after combination

Figure 3: The basic logical query plan of a Pregel superstep i which reads the data generated from the last superstep (e.g., $\texttt{Vertex}_i$, $\texttt{Msg}_i$, and $\texttt{GS}_i$) and produces the data (e.g., $\texttt{Vertex}_{i+1}$, $\texttt{Msg}_{i+1}$, and $\texttt{GS}_{i+1}$) for superstep $i+1$. Global aggregation and synchronization are in Figure 4, and vertex addition and removal are in Figure 5.



Flow	Data
D4	The global halting state contribution
D5	Values for *aggregate*
D8	The global halt state
D9	The global aggregate value
D10	The increased superstep

Figure 4: The plan segment that revises the global state.



Flow	Data
D6	Vertex tuples for deletions and insertions

Figure 5: The plan segment for vertex addition/removal.

In datafow $D3$, the message output is grouped by destination vertex id and aggregated by the `combine` function[4], which produces flow $D7$ that is inserted into the `Msg` relation.

The global state relation `GS` contains a single tuple whose fields comprise the global state. Figure 4 describes the flows that revise these fields in each superstep. The halt state and global aggregate fields depend on the output of `compute`, while the superstep counter is simply its previous value plus one. Flow $D4$ applies a boolean aggregate function (logical AND) to the global halting state contribution from each vertex; the output (flow $D8$) indicates the global halt state, which controls the execution of another superstep. Flow $D5$ routes the global aggregate state contributions from all active vertices to the `aggregate` UDF which then produces the global aggregate value (flow $D9$) for the next superstep.

Graph mutations are specified by a `Vertex` tuple with an operation that indicates insertion (adding a new vertex) or deletion (removing a vertex)[5]. Flow $D6$ in Figure 5 groups these mutation tuples by vertex id and applies the `resolve` function to each group. The output is then applied to the `Vertex` relation.

[4]The default `combine` gathers all messages for a given destination into a list.

[5]Pregelix leaves the application-specific vertex deletion semantics in terms of integrity constraints to application programmers.

4. THE RUNTIME IMPLEMENTATION

The Pregel logical plan could be implemented on any parallel dataflow engine, including Stratosphere [11], Spark [44] or Hyracks [12]. As we argue below, we believe that Hyracks is particularly well-suited for this style of computation; this belief is supported by Section 7's experimental results (where some of the systems studied are based on other platforms). The rest of this section covers the Hyracks platform [12], which is Pregelix's target runtime for the logical plan in Section 3. Hyracks is a data-parallel runtime in the same general space as Hadoop [5] and Dryad [30]. Jobs are submitted to Hyracks in the form of DAGs (directed acyclic graphs) that are made up of operators and connectors. Operators are responsible for consuming input partitions and producing output partitions. Connectors perform redistributions of data between operators. For a submitted job, in a Hyracks cluster, a master machine dictates a set of worker machines to execute clones of the operator DAG in parallel and orchestrates data exchanges. Below, we enumerate the features and components of Hyracks that we leverage to implement the logical plan described in Section 3.

User-configurable task scheduling. The Hyracks engine allows users to express task scheduling constraints (e.g., count constraints, location choice constraints, or absolute location constraints) for each physical operator. The task scheduler of Hyracks is a constraint solver that comes up with a schedule satisfying the user-defined constraints. In Section 5.3.4, we leverage this feature of Hyracks to implement sticky, iterative dataflows.

Access methods. B-trees and LSM B-trees are part of the Hyracks storage library. A B-tree [19] is a commonly used index structure in most commercial databases; it supports efficient lookup and scan operations, but a single tree update can cause random I/Os. In contrast, the LSM B-tree [34] puts updates into an in-memory component (e.g., an in-memory B-tree); it merges the in-memory component with disk components in a periodic manner, which turns random I/Os for updates into sequential ones. The LSM B-tree thus allows fast updates but may result in slightly slower lookups.

Group-by operators. The Hyracks operator library includes three group-by operator implementations: sort-based group-by, which pushes group-by aggregations into both the in-memory sort phase and the merge phase of an external sort operator; HashSort group-by, which does the same thing as the sort-based one except using hash-based group-by for the in-memory processing phase; and preclustered group-by, which assumes incoming tuples are already clustered by the group-by key and hence just applies the aggregation operation in sequence to one group after the other.

Join operators. Merge-based index full outer join and probe-based index left-outer join are supported in the Hyracks operator library. The full outer join operator merges sorted input from the outer relation with an index scan on the inner relation; tuples containing `NULL` values for missing fields will be generated for no-matches. The left-outer join operator, for each input tuple in the outer relation, consults an index on the inner relation for matches that produce join results or for no-matches that produce tuples with `NULL` values for the inner relation attributes.

Connectors. Hyracks connectors define inter-operator data exchange patterns. Here, we focus on the following three communication patterns: an m-to-n partitioning connector repartitions the data based on a user-defined partitioning function from m (sender-side) partitions to n (receiver-side) partitions; the m-to-n partitioning merging connector does the same thing but assumes tuples from the sender-side are ordered and therefore simply merges the input streams at the receiver-side; the aggregator connector reduces all input streams to a single receiver partition.

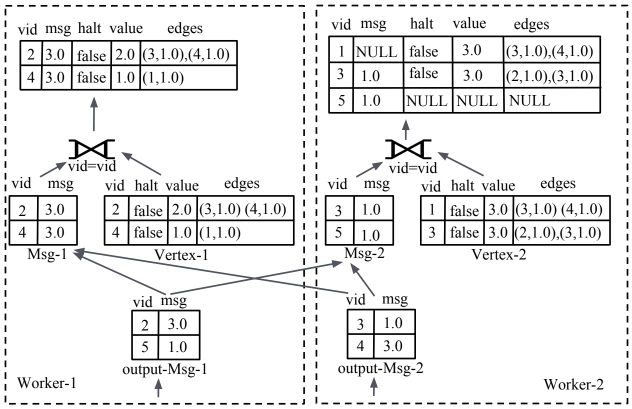


Figure 6: The parallelized join for the logical join in Figure 2.

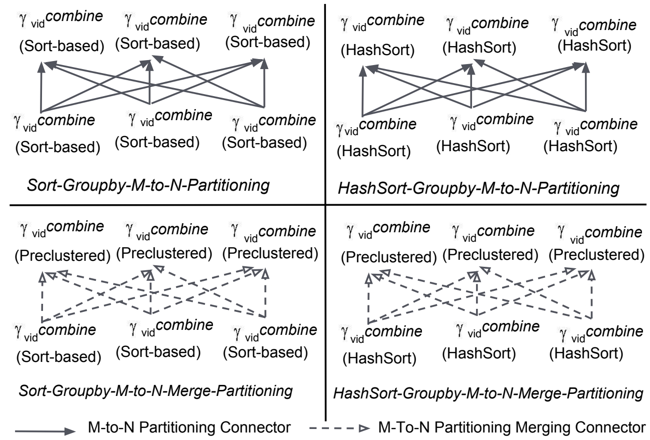


Figure 7: The four physical group-by strategies for the group-by operator which combines messages in Figure 3.

Materialization policies. We use two materialization policies that Hyracks supports for customizing connectors: fully pipelined, where the data from a producer is immediately pushed to the consumer, and sender-side materializing pipelined, where the data transfer channel launches two threads at the sender side, one that writes output data to a local temporary file, and another that pulls written data from the file and sends it to the receiver-side.

5. THE PREGELIX SYSTEM

In this section, we describe our implementation of the logical plan (Section 3) on the Hyracks runtime (Section 4), which is core of the Pregelix system. We elaborate on data-parallel execution (Section 5.1), data storage (Section 5.2), physical query plan alternatives (Section 5.3), memory management (Section 5.4), fault-tolerance (Section 5.5), and job pipelining (Section 5.6). We conclude by summarizing the software components of Pregelix (Section 5.7) and revisiting our three opportunities (Section 5.8).

5.1 Parallelism

To parallelize the logical plan of the Pregel computation described in Section 3 at runtime, one or more clones of a physical plan—that implements the logical plan—are shipped to Hyracks worker machines that run in parallel. Each clone deals with a single data partition. During execution, data is exchanged from the clones of an upstream operator to those of a downstream operator through a Hyracks connector. Figure 6 shows an example, where the logical join described in Figure 2 is parallelized onto two workers and message tuples are exchanged from producer partitions (operator clones) to consumer partitions (operator clones) using an m-to-n partitioning connector, where m and n are equal to two.

5.2 Data Storage

Given a graph analytical job, Pregelix first loads the input graph dataset (the initial `Vertex` relation) from a distributed file system, i.e., HDFS, into a Hyracks cluster, partitioning it by *vid* using a user-defined partitioning function[6] across the worker machines. After the eventual completion of the overall Pregel computation, the partitioned `Vertex` relation is scanned and dumped back to HDFS. During the supersteps, at each worker node, one (or more) local indexes—keyed off of the *vid* field—are used to store one (or more) partitions of the `Vertex` relation. Pregelix leverages both B-tree and LSM B-tree index structures from the Hyracks storage library to store partitions of `Vertex` on worker machines. The choice of which index structure to use is workload-dependent and user-selectable. A B-tree index performs well on jobs that frequently update vertex data in-place, e.g., PageRank. An LSM B-tree index performs well when the size of vertex data is changed drastically from superstep to superstep, or when the algorithm performs frequent graph mutations, e.g., the path merging algorithm in genome assemblers [45].

The `Msg` relation is initially empty; it is refreshed at the end of a superstep with the result of the message `combine` function call in the (logical) dataflow $D7$ of Figure 3; the physical plan is described in Section 5.3.1. The message data is partitioned by destination vertex id (*vid*) using the same partitioning function applied to the vertex data, and is thus stored (in temporary local files) on worker nodes that maintain the destination vertex data. Furthermore, each message partition is sorted by the *vid* field.

Lastly, we leverage HDFS to store the global state of a Pregelix job; an access method is used to read and cache the global state at worker nodes when it is referenced by user-defined functions like `compute`.

5.3 Physical Query Plans

In this subsection, we dive into the details of the physical plans for the logical plan described in Figures 3, 4, and 5. Our discussion will cover message combination and delivery, global states, graph mutations, and data redistribution.

5.3.1 *Message Combination*

Figure 3 uses a logical group-by operator for message combination. For that, Pregelix leverages the three group-by operator implementations mentioned in Section 4. A preclustered group-by can only be applied to input data that is already clustered by the grouping key. A HashSort group-by operator offers better performance (over sort-based group-by) when the number of groups (the number of distinct message receivers in our case) is small; otherwise, these two group-by operators perform similarly. In a parallel execution, the grouping is done by two stages—each producer partitions its output (message) data by destination *vid*, and the output is redistributed (according to destination *vid*) to each consumer, which performs the final grouping step.

Pregelix has four different parallel group-by strategies, as shown in Figure 7. The lower two strategies use an m-to-n partitioning merging connector and only need a simple one-pass pre-clustered group-by at the receiver-side; however, in this case, receiver-side merging needs to coordinate the input streams, which takes more

[6]By default, we use hash partitioning.

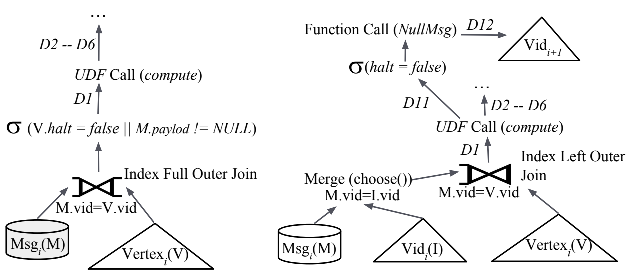


Figure 8: Two physical join strategies for forming the input to the `compute` UDF. On the left is an index full outer join approach. On the right is an index left outer join approach.

time as the cluster size grows. The upper two strategies use an m-to-n partitioning connector, which does not require such coordination; however, these strategies do not deliver sorted data streams to the receiver-side group-bys, so re-grouping is needed at the receiver-side. A fully pipelined policy is used for the m-to-n partitioning connector in the upper two strategies, while in the lower two strategies, a sender-side materializing pipelined policy is used by the m-to-n partitioning merging connector to avoid possible deadlock scenarios mentioned in the query scheduling literature [27]. The choice of which group-by strategy to use depends on the dataset, graph algorithm, and cluster. We will further pursue this choice in our experiments (Section 7).

5.3.2 Message Delivery

Recall that in Figure 3, a logical full-outer-join is used to deliver messages to the right vertices and form the data needed to call the `compute` UDF. For that, we use index-based joins because (a) vertices are already indexed by vid, and (b) all four group-by strategies in Figure 7 flow the (combined) messages out of their receiver-side group-bys in vid-sorted order, thereby producing vid-sorted Msg partitions.

Pregelix offers two physical choices for index-based joins—an index full outer join approach and an index left outer join approach, as shown in Figure 8. The full outer join plan scans the entire vertex index to merge it with the (combined) messages. This join strategy is suitable for algorithms where most vertices are live (active) across supersteps (e.g., PageRank). The left outer join plan prunes unnecessary vertex scans by first searching the live vertex index for each (combined) incoming message, and it fits cases where messages are sparse and only few vertices are live in every superstep (e.g., single source shortest paths). A user can control which join approach Pregelix uses for a given job. We now briefly explain the details of the two join approaches.

Index Full Outer Join. As shown in left side of Figure 8, this plan is straightforwardly mapped from the logical plan. The join operator simply merges a partition of `Msg` and `Vertex` using a single pass.

Index Left Outer Join. As shown in right of Figure 8, this plan initially bulk loads another B-tree `Vid` with null messages (vid, NULL) that are generated by a function `NullMsg`. This index serves to represent the set of currently active vertices. The dataflows $D11$ and $D12$ in Figure 8 are (vid, halt) tuples and (vid, NULL) tuples respectively. Note that `Vid` is partitioned in the same way as `Vertex`. In the next superstep, a merge operator merges tuples from `Msg` and `Vid` based on the equivalence of the `vid` fields, and the *choose* function inside the operator selects tuples from `Msg` to output when there are duplicates. Output tuples of the merge operator are sent to an index left outer join operator that probes the `Vertex` index. Tuples produced by the left outer join operator are directly sent to the `compute` UDF. The original filter operator σ(*V.halt=false*||*M.payload!= NULL*) in the logical plan is logically transformed to the merge operator where tuples in `Vid` satisfy *halt=false* and tuples in `Msg` satisfy *M.payload!= NULL*.

To minimize data movements among operators, in a physical plan, we push the filter operator, the UDF call of `compute`, the update to `Vertex`, and the extraction (project) of fields in the output tuple of `compute` into the upstream join operator as Hyracks "mini-operators."

5.3.3 Global States and Graph Mutations

To form the global halt state and aggregate state—see the two global aggregations in Figure 4—we leverage a standard two-stage aggregation strategy. Each worker pre-aggregates these state values (stage one) and sends the result to a global aggregator that produces the final result and writes it to HDFS (stage two). The incrementing of superstep is also done by a trivial dataflow.

The additions and removals of vertices in Figure 5 are applied to the `Vertex` relation by an index insert-delete operator. For the group-by operator in Figure 5, we only do a receiver-side group-by because the `resolve` function is not guaranteed to be distributive and the connector for $D6$ (in Figure 3) is an m-to-n partitioning connector in the physical plan.

5.3.4 Data Redistribution

In a physical query plan, data redistribution is achieved by either the m-to-n hash partitioning connector or the m-to-n hash partitioning merging connector (mentioned in Section 4). With the Hyracks provided user-configurable scheduling, we let the location constraints of the join operator (in Figure 8) be the same as the places where partitions of `Vertex` are stored across all the supersteps. Also, the group-by operator (in Figure 7) has the same location constraints as the join operator, such that in all supersteps, `Msg` and `Vertex` are partitioned in the same (sticky) way and the join between them can be done without extra repartitioning. Therefore, the only necessary data redistributions in a superstep are (a) redistributing outgoing (combined) messages from sender partitions to the right vertex partitions, and (b) sending each vertex mutation to the right partition for addition or removal in the graph data.

5.4 Memory Management

Hyracks operators and access methods already provide support for out-of-core computations. The default Hyracks memory parameters work for all aggregated memory sizes as long as there is sufficient disk space on the worker machines. To support both in-memory and out-of-core workloads, B-trees and LSM-trees both leverage a buffer cache that caches partition pages and gracefully spills to disk only when necessary using a standard replacement policy, i.e., LRU. In the case of an LSM B-tree, some number of buffer pages are pinned in memory to hold memory-resident B-tree components.

The majority of the physical memory on a worker machine is divided into four parts: the buffer cache for access methods of the `Vertex` relation; the buffers for the group-by operator clones; the buffers for network channels; and the file system cache for (a) temporary run files generated by group-by operator clones, (b) temporary files for materialized data redistributions, and (c) temporary files for the relation `Msg`. The first three memory components are explicitly controlled by Pregelix and can be tuned by a user, while the last component is (implicitly) managed by the underlying OS. Although the Hyracks runtime is written in Java, it uses a bloat-aware design [14] to avoid unnecessary memory bloat and to minimize the performance impact of garbage collection in the JVM.

5.5 Fault-Tolerance

Pregelix offers the same level of fault-tolerance as other Pregel-like systems [32, 4, 6] by checkpointing states to HDFS at user-selected superstep boundaries. In our case, the states to be checkpointed at the end of a superstep include `Vertex` and `Msg` (as well as `Vid` if the left outer join approach is used). The checkpointing of `Msg` ensures that a user program does not need to be aware of failures. Since `GS` stores its primary copy in HDFS, it need not be part of the checkpoint. A user job can determine whether or not to checkpoint after a superstep. Once a node failure or disk failure happens, the failed machine is added into a blacklist.

During recovery, Pregelix finds the latest checkpoint and reloads the states to a newly selected set of failure-free worker machines. Reloading states includes two steps. First, it kicks off physical query plans to scan, partition, sort, and bulk load the entire `Vertex` and `Vid` (if any) from the checkpoint into B-trees (or LSM B-trees), one per partition. Second, it executes another physical query plan to scan, partition, sort, and write the checkpointed `Msg` data to each partition as a local file.

5.6 Job Pipelining

Pregelix can accept an array of jobs and pipeline between compatible contiguous jobs without HDFS writes/reads nor index bulk-loads. Two compatible jobs should have a producer-consumer relationship regarding the output/input data and share the same type of vertex—meaning, they interpret the corresponding bits in the same way. This feature was motivated by the genome assembler [45] application which runs six different graph cleaning algorithms that are chained together for many rounds. A user can choose to enable this option to get improved performance with reduced fault-tolerance.

5.7 Pregelix Software Components

Pregelix supports the Pregel API introduced in Section 2.1 in Java, which is very similar to the APIs of Giraph [4] and Hama [6]. Internally, Pregelix has a statistics collector, failure manager, scheduler, and plan generator which run on a client machine after a job is submitted; it also has a runtime context that stays on each worker machine. We describe each component below.

Statistics Collector. The statistics collector periodically collects statistics from the target Hyracks cluster, including system-wide counters such as CPU load, memory consumption, I/O rate, network usage of each worker machine, and the live machine set, as well as Pregel-specific statistics such as the vertex count, live vertex count, edge count, and message count of a submitted job.

Failure Manager. The failure manager analyzes failure traces and recovers from those failures that are indeed recoverable. It only tries to recover from interruption errors (e.g., a worker machine is powered off) and I/O related failures (e.g., disk I/O errors); it just forwards application exceptions to end users. Recovery is done as mentioned in Section 5.5.

Scheduler. Based on the information obtained by the statistics collector and the failure manager, the scheduler determines which worker machines to use to run a given job. The scheduler assigns as many partitions to a selected machine as the number of its cores. For each Pregel superstep, Pregelix sets location constraints for operators in the manner mentioned in Section 5.3.4. For loading `Vertex` from HDFS [5], the constraints of the data scanning operator (set by the scheduler) exploit data locality for efficiency.

Plan Generator. The plan generator generates physical query plans for data loading, result writing, each single Pregel superstep, checkpointing, and recovery. The generated plan includes a physical operator DAG and a set of location constraints for each operator.

Runtime Context. The runtime context stores the cached `GS` tuple and maintains the Pregelix-specific implementations of the Hyracks extensible hooks to customize buffer, file, and index management.

5.8 Discussion

Let us close this section by revisiting the issues and opportunities presented in Section 2.3 and evaluating their implications in Pregelix:

- Out-of-core Support. All the data processing operators as well as access methods we use have out-of-core support, which allows the physical query plans on top to be able to run disk-based workloads as well as multi-user workloads while retaining good in-memory processing performance.
- Physical Flexibility. The current physical choices spanning vertex storage (two options), message delivery (two alternatives), and message combination (four strategies) allow Pregelix to have sixteen ($2 \times 2 \times 4$) tailored executions for different kinds of datasets, graph algorithms, and clusters.
- Software Simplicity. The implementations of all the described functionalities in this section leverage existing operator, connector, and access method libraries provided by Hyracks. Pregelix does not involve modifications to the Hyracks runtime.

6. PREGELIX CASE STUDIES

In this section, we briefly enumerate several Pregelix use cases, including a built-in graph algorithm library, a study of graph connectivity problems, and research on parallel genome assembly.

The Pregelix Built-in Library. The Pregelix software distribution comes with a library that includes several graph algorithms such as PageRank, single source shortest paths, connected components, reachability query, triangle counting, maximal cliques, and random-walk-based graph sampling. Figure 9 shows the single source shortest paths implementation on Pregelix, where hints for the join, group-by, and connector choices are set in the `main` function. Inside `compute`, the method calls to set a vertex value and to send a message internally generate output tuples for the corresponding dataflows.

Graph Connectivity Problems. Using Pregelix, a graph analytics research group in Hong Kong has implemented several graph algorithm building blocks such as BFS (breath first search) spanning tree, Euler tour, list ranking, and pre/post-ordering. These building blocks have been used to develop advanced graph algorithms such as bi-connected components for undirected graphs (e.g., road networks) and strongly connected components for directed graphs (e.g., the Twitter follower network) [42] . The group also scale-tested all of their algorithms on a 60 machine cluster with 480 cores and 240 disks, using Pregelix as the infrastructure.

Genome Assembly. Genomix [3] is a data-parallel genome assembler built on top of Pregelix. It first constructs a (very large) De Bruijn graph [45] from billions of genome reads, and then (a) cleans the graph with a set of pre-defined subgraph patterns (described in [45]) and (b) merges available single paths into vertices iteratively until all vertices can be merged to a single (gigantic) genome sequence. Pregelix's support for the addition and removal of vertices is heavily used in this use case.

7. EXPERIMENTS

This section compares Pregelix with several other popular parallel graph processing systems, including Giraph [4], Hama [6],

```
public class ShortestPathsVertex extends Vertex<VLongWritable, DoubleWritable,
            FloatWritable, DoubleWritable> {
  /** The source id key */
  public static final String SOURCE_ID = "pregelix.sssp.sourceId";
  /** The value to be sent to neighbors*/
  private DoubleWritable outputValue = new DoubleWritable();
  /** the source vertex id */
  private long sourceId = -1;

  @Override
  public void configure(Configuration conf) {
    sourceId = conf.getLong(SOURCE_ID, 1);
  }

  @Override
  public void compute(Iterator<DoubleWritable> msgIterator) {
    if (getSuperstep() == 1) {
      getVertexValue().set(Double.MAX_VALUE);
    }
    double minDist = getVertexId().get() == sourceId? 0.0 : Double.MAX_VALUE;
    while (msgIterator.hasNext()) {
      minDist = Math.min(minDist, msgIterator.next().get());
    }
    if (minDist < getVertexValue().get()) {
      getVertexValue().set((minDist);
      for (Edge<VLongWritable, FloatWritable> edge : getEdges()) {
        outputValue.set(minDist + edge.getEdgeValue().get());
        sendMsg(edge.getDestVertexId(), outputValue);
      }
    }
    voteToHalt();
  }

  public static void main(String[] args) throws Exception {
    PregelixJob job = new PregelixJob(ShortestPathsVertex.class.getSimpleName());
    job.setVertexClass(ShortestPathsVertex.class);
    job.setVertexInputFormatClass(SimpleTextInputFormat.class);
    job.setVertexOutputFormatClass(SimpleTextOutputFormat.class);
    job.setMessageCombinerClass(DoubleMinCombiner.class);

    /** Hints for the Pregelix plan generator */
    job.setMessageVertexJoin(Join.LEFTOUTER);
    job.setMessageGroupBy(GroupBy.HASHSORT);
    job.setMessageGroupByConnector(Connector.UNMERGE);

    Client.run(args, job);
  }
}
```

Figure 9: The implementation of the single source shortest paths algorithm on Pregelix.

GraphLab [31], and GraphX [40]. Our comparisons cover execution time (Section 7.2), scalability (Section 7.3), throughput (Section 7.4), plan flexibility (Section 7.5), and software simplicity (Section 7.6). We conclude this section by summerizing our experimental results (Section 7.7).

7.1 Experimental Setup

We ran the experiments detailed here on a 32-node Linux IBM x3650 cluster with one additional master machine of the same configuration. Nodes are connected with a Gigabit Ethernet switch. Each node has one Intel Xeon processor E5520 2.26GHz with four cores, 8GB of RAM, and two 1TB, 7.2K RPM hard disks.

In our experiments, we leverage two real-world graph-based datasets. The first is the Webmap dataset [41] taken from a crawl of the web in the year 2002. The second is the BTC dataset [18], which is a undirected semantic graph converted from the original Billion Triple Challenge 2009 RDF dataset [2]. Table 3 (Webmap) and Table 4 (BTC) show statistics for these graph datasets, including the full datasets as well as several down-samples and scale-ups[7] that we use in our experiments.

Our evaluation examines the platforms' performance characteristics of three algorithms: PageRank [35], SSSP (single source shortest paths) [23], and CC (connected components) [23]. On Pregelix and Giraph, the graph algorithms were coded in Java and all their source code can be found in the Pregelix codebase[8]. The implementations of the three algorithms on Hama, GraphLab, and GraphX are directly borrowed from their builtin examples. The Pregelix default plan, which uses index full outer join, sort-based group-by, an m-to-n hash partitioning connector, and B-tree vertex storage, is used in Sections 7.2, 7.3, and 7.4. Pregelix's default maximum buffer cache size for access methods is set to $\frac{1}{4}$ the physical RAM size, and its default maximum allowed buffer for each group-by operator instance is set to 64MB. These two default Pregelix memory settings are used in all the experiments. For the local file system for Pregelix, we use the ext3 file system; for the distributed file system, we use HDFS version 1.0.4. In all experiments, we use the latest Giraph trunk version (the revision at Aug 26 11:35:14 2014), Hama version 0.6.4, GraphLab version 2.2 (PowerGraph), and Spark [44] version 0.9.1 for GraphX. Our GraphLab setting has been confirmed by its primary author. We tried our best to let each system use all the CPU cores and all available RAM on each worker machine.

Name	Size	#Vertices	#Edges	Avg. Degree
Large	71.82GB	1,413,511,390	8,050,112,169	5.69
Medium	31.78GB	709,673,622	2,947,603,924	4.15
Small	14.05GB	143,060,913	1,470,129,872	10.27
X-Small	9.99GB	75,605,388	1,082,093,483	14.31
Tiny	2.93GB	25,370,077	318,823,779	12.02

Table 3: The Webmap dataset (Large) and its samples.

Name	Size	#Vertices	#Edges	Avg. Degree
Large	66.48GB	690,621,916	6,177,086,016	8.94
Medium	49.86GB	517,966,437	4,632,814,512	8.94
Small	33.24GB	345,310,958	3,088,543,008	8.94
X-Small	16.62GB	172,655,479	1,544,271,504	8.94
Tiny	7.04GB	107,706,280	607,509,766	5.64

Table 4: The BTC dataset (X-Small) and its samples/scale-ups.

7.2 Execution Time

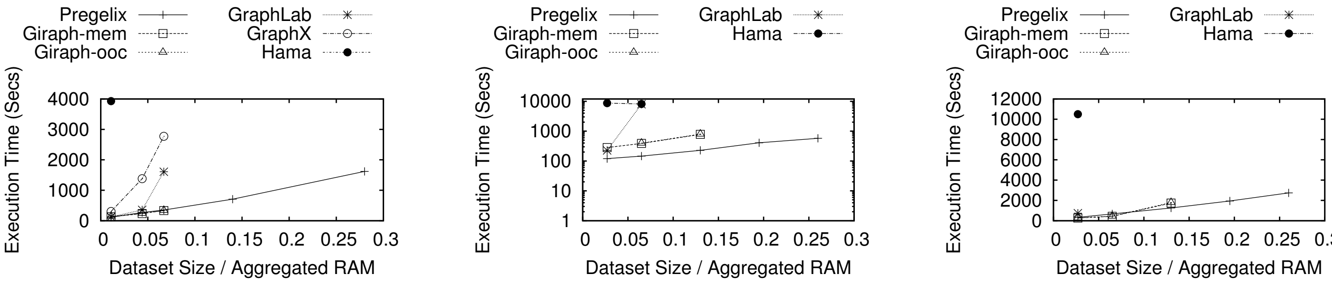
In this experiment, we evaluate the execution times of all systems by running each of the three algorithms on the 32-machine cluster. As the input data for PageRank we use the Webmap dataset because PageRank is designed for ranking web pages, and for the SSSP and CC algorithms we use the BTC dataset. Since a Giraph user needs to explicitly specify apriori whether a job is in-memory or out-of-core, we measure both of these settings (labeled Giraph-mem and Giraph-ooc, respectively) for Giraph jobs regardless of the RAM size.

Figure 10 plots the resulting *overall* Pregel job execution times for the different sized datasets, and Figure 11 shows the average *per-iteration* execution time for all iterations. In both figures, the x-axis is the input dataset size relative to the cluster's aggregated RAM size, and the y-axis is the execution time. (Note that the volume of exchanged messages can exhaust memory even if the initial input graph dataset can fit into memory.)

Figure 10 and Figure 11 show that while Pregelix scales to out-of-core workloads, Giraph fails to run the three algorithms once the relative dataset size exceeds 0.15, even with its out-of-core setting enabled. Figures 10(a)(c) and 11(a)(c) show that when the computation has sufficient memory, Pregelix offers comparable execution time to Giraph for message-intensive workloads such as PageRank and CC. For PageRank, Pregelix runs up to 2× slower than Giraph on relatively small datasets like Webmap-Small. For CC, Pregelix runs up to 1.7× faster than Giraph on relatively small datasets like BTC-Small. Figure 10(b) and Figure 11(b) demonstrate that

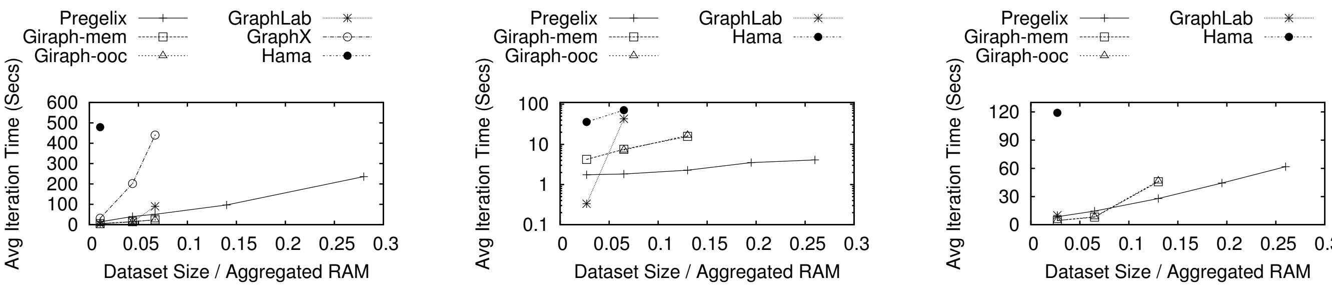
[7] We used a random walk graph sampler built on top of Pregelix to create scaled-down Webmap sample graphs of different sizes. To scale up the BTC data size, we deeply copied the original graph data and renumbered the duplicate vertices with a new set of identifiers.

[8] https://code.google.com/p/hyracks/source/browse/pregelix



(a) PageRank for Webmap datasets (b) SSSP for BTC datasets (c) CC for BTC datasets

Figure 10: Overall execution time (32-machine cluster). Neither Giraph-mem nor Giraph-ooc can work properly (both keep failing and re-running tasks) when the ratio of dataset size to the aggregated RAM size exceeds 0.15; GraphLab starts failing when the ratio of dataset size to the aggregated RAM size exceeds 0.07; Hama fails on even smaller datasets; GraphX keeps failing because of memory management issues when it runs over the smallest BTC dataset sample BTC-Tiny.



(a) PageRank for Webmap datasets (b) SSSP for BTC datasets (c) CC for BTC datasets

Figure 11: Average iteration execution time (32-machine cluster).

the Pregelix default plan offers $3.5\times$ overall speedup and $7\times$ per-iteration speedup over Giraph for message-sparse workloads like SSSP even for relatively small datasets. All sub-figures in Figure 10 and Figure 11 show that for in-memory workloads (when the relative dataset size is less than 0.15), Giraph has steeper (worse) size-scaling curves than Pregelix.

Compared to Giraph, GraphLab, GraphX, and Hama start failing on even smaller datasets, with even steeper size-scaling curves. GraphLab has the best average per-iteration execution time on small datasets (e.g., up to $5\times$ faster than Pregelix and up to $12\times$ faster than Giraph, on BTC-Tiny), but performs worse than Giraph and Pregelix on larger datasets (e.g., up to $24\times$ slower than Pregelix and up to $6\times$ slower than Giraph, on BTC-X-Small). GraphX keeps failing because of memory management issues during processing the smallest BTC dataset BTC-Tiny for SSSP and CC on the 32-machine cluster, thus its results for both algorithms are missing.

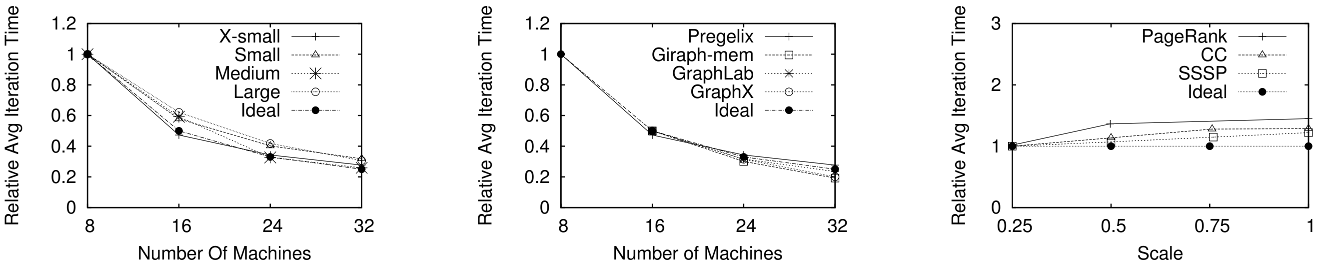
7.3 System Scalability

Our system scalability experiments run the different systems on varying sized clusters for each of the different dataset sizes. Figure 12(a) plots the parallel speedup for PageRank on Pregelix going from 8 machines to 32 machines. The x-axis is the number of machines, and the y-axis is the average per-iteration execution time relative to the time on 8 machines. As the number of machines increases, the message combiner for PageRank becomes less effective and hence the total volume of data transferred through the network gets larger, though the CPU load of each individual machine drops. Therefore, in Figure 12(a), the parallel speedups are close to but slightly worse than the "ideal" case in which there are no message overheads among machines. For the other systems, the PageRank implementations for GraphLab and GraphX fail to run Webmap samples beyond the 9.99GB case when the number of machines is 16; Giraph has the same issue when the number of machines is 8. Thus, we were only able to compare the parallel PageRank speedups of Giraph, GraphLab, GraphX, and Pregelix with the Webmap-X-Small dataset. The results of this small data case are in Figure 12(b); Hama is not included because it cannot run even the Webmap-X-Small dataset on any of our cluster configurations. The parallel speedup of Pregelix is very close to the ideal line, while Giraph, GraphLab, and GraphX exhibit even better speedups than the ideal. The apparent super-linear parallel speedups of Giraph, GraphLab, and GraphX are consistent with the fact that they all perform super-linearly worse when the volume of data assigned to a slave machine increases, as can be seen in Figures 10 and 11.

Figure 12(c) shows the parallel scale up of the three algorithms for Pregelix. Giraph, GraphLab, GraphX, and Hama results are not shown because they cannot successfully run these cases. In this figure, the x-axis is the ratio of the sampled (or scaled) dataset size over the largest (Webmap-Large or BTC-Large) dataset size. The number of machines is proportional to this ratio and 32 machines are used for scale 1.0. The y-axis is the average per-iteration execution time relative to the time at the smallest scale. In the ideal case, the y-axis value would stay at 1.0 for all the scales, while in reality, the three Pregel graph algorithms all incur network communication and thus cannot achieve the ideal. The SSSP algorithm sends fewer messages than the other two algorithms, so it is the closest to the ideal case.

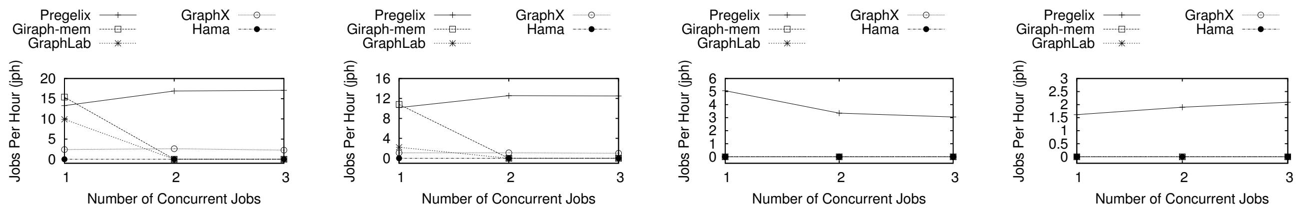
7.4 Throughput

In the current version of GraphLab, Hama, and Pregelix, each submitted job is executed immediately, regardless of the current activity level on the cluster. Giraph leverages Hadoop's admission control; for the purpose of testing concurrent workloads, we let the number of Hadoop map task slots in each task tracker be 3. GraphX leverages the admission control of Spark, such that jobs will be executed sequentially if available resources cannot meet the



(a) Pregelix Speedup (PageRank) (b) Speedup (PageRank) for Webmap-X-Small (c) Pregelix Scaleup (PageRank, SSSP, CC)

Figure 12: Scalability (run on 8-machine, 16-machine, 24-machine, 32-machine clusters).



(a) Webmap-X-Small (always in-memory) (b) Webmap-Small (in-memory to minor disk usage) (c) Webmap-Medium (in-memory to disk-based) (d) Webmap-Large (always disk-based)

Figure 13: Throughput (multiple PageRank jobs are executed in the 32-machine cluster with different sized datasets).

overall requirements of concurrent jobs. In this experiment, we compare the job throughput of all the systems by submitting jobs concurrently. We ran PageRank jobs on the 32-machine cluster using four different samples of the Webmap dataset (X-Small, Small, Medium, and Large) with various levels of job concurrency. Figure 13 reports how the number of completed jobs per hour (**jph**) changes with the number of concurrent jobs. The results for the four Webmap samples represent four different cases respectively:

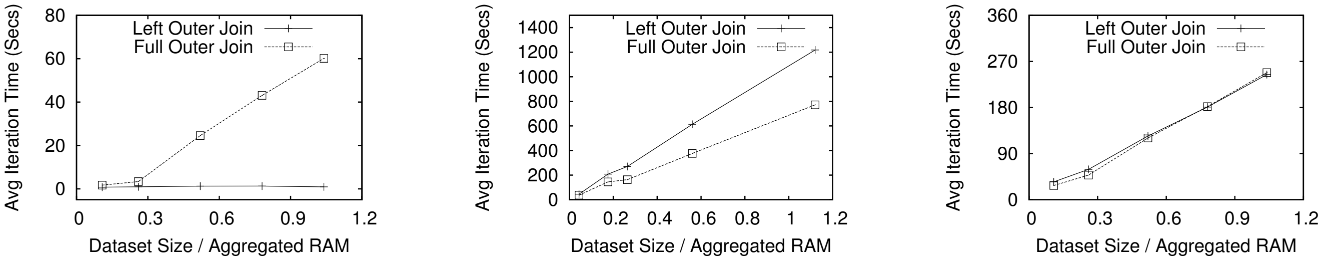
- Figure 13(a) uses Webmap-X-Small. Moving from serial job execution to concurrent job execution, data processing remains in-memory but the CPU resources go from dedicated to shared. In this case, Pregelix achieves a higher jph when there are two or three concurrent jobs than when job execution is serial.
- Figure 13(b) uses Webmap-Small. In this case, serial job execution does in-memory processing, but concurrent job execution introduces a small amount of disk I/O due to spilling. When two jobs run concurrently, each job incurs about 1GB of I/O; when three jobs run concurrently, each does about 2.7GB of I/O. In this situation, still, the Pregelix jph is higher in concurrent execution than in serial execution.
- Figure 13(c) uses Webmap-Medium. In this case, serial job execution allows for in-memory processing, but allowing concurrent execution exhausts memory and causes a significant amount of I/O for each individual job. For example, when two jobs run concurrently, each job incurs about 10GB of I/O; when three jobs run concurrently, each job does about 27GB of I/O. In this case, jph drops significantly at the boundary where I/O significantly comes into the picture. The point with two concurrent jobs is such a boundary for Pregelix in Figure 13(c).
- Figure 13(d) uses the full Webmap (Webmap-Large). In this case, processing is always disk-based regardless of the concurrency level. For this case, Pregelix jph once again increases with the increased level of concurrency; this is because the CPU utilization is increased (by about 20% to 30%) with added concurrency.

These results suggest that it would be worthwhile to develop intelligent job admission control policies to make sure that Pregelix runs with the highest possible throughput all the time in our future work. In our experiments, the Spark scheduler for GraphX always runs concurrent jobs sequentially due to the lack of memory and CPU resources. Giraph, GraphLab, and Hama all failed to support concurrent jobs in our experiments for all four cases due to limitations regarding memory management and out-of-core support; they each need additional work to operate in this region.

7.5 Plan Flexibility

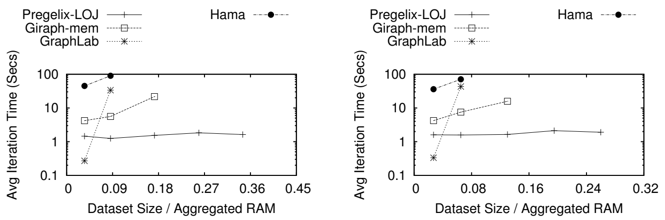
In our final experiment, we compare several different physical plan choices in Pregelix to demonstrate their usefulness. We ran the two join plans (described in Section 5.3.2) for the three Pregel graph algorithms. Figure 14 shows the results. For message-sparse algorithms like SSSP (Figure 14(a)), the left outer join Pregelix plan is much faster than the (default) full outer join plan. However, for message-intensive algorithms like PageRank (Figure 14(b)), the full outer join plan is the winner. This is because although the probe-based left outer join can avoid a sequential index scan, it needs to search the index from the root node every time; this is not worthwhile if most data in the leaf nodes will be qualified as join results. The CC algorithm's execution starts with many messages, but the message volume decreases significantly in its last few supersteps, and hence the two join plans result in similar performance (Figure 14(c)). Figure 15 revisits the relative performance of the systems by comparing Pregelix's left outer join plan performance against the other systems. As shown in the figure, SSSP on Pregelix can be up to 15× faster than on Giraph and up to 35× faster than on GraphLab for the average per-iteration execution time.

In addition to the experiments presented here, an earlier technical report [13] measured the performance difference introduced by the two different Pregelix data redistribution policies (as described in Section 5.3.1) for combining messages on a 146-machine cluster in Yahoo! Research. Figure 9 in the report [13] shows that the m-to-n hash partitioning merging connector can lead to slightly faster executions on small clusters, but merging input streams at the receiver side needs to selectively wait for data to arrive from specific senders as dictated by the priority queue, and hence it becomes slower on larger clusters. The tradeoffs seen here and in the technical report [13] for different physical choices are evidence that an



(a) SSSP for BTC datasets (b) PageRank for Webmap datasets (c) CC for BTC datasets

Figure 14: Index full outer join vs. index left outer join (run on an 8 machine cluster) for Pregelix.



(a) 24-Machine Cluster (b) 32-Machine Cluster

Figure 15: Pregelix left outer join plan vs. other systems (SSSP on BTC datasets). GraphX hungs because of memory management issues when it runs over the smallest dataset.

optimizer is ultimately essential to identify the best physical plan to use in order to efficiently execute Pregel programs.

7.6 Software Simplicity

To highlight the benefit of building a Pregel implementation on top of an existing dataflow runtime, as opposed to writing one from scratch, we can compare lines of code in the Pregelix and Giraph systems' core modules (excluding their test code and comments). The Giraph-core module, which implements the Giraph infrastructure, contains 32,197 lines of code. Its counterpart in Pregelix contains just 8,514 lines of code.

7.7 Summary

Our experimental results show that Pregelix can perform comparably to Giraph for memory-resident message-intensive workloads (like PageRank), can outperform Giraph by over an order of magnitude for memory-resident message-sparse workloads (like single source shortest paths), can scale to larger datasets, can sustain multi-user workloads, and also can outperform GraphLab, GraphX, and Hama by over an order of magnitude on large datasets for various workloads. In addition to the numbers reported in this paper, an early technical report [13] gave speedup and scale-up results for the first alpha release of Pregelix on a 146-machine Yahoo! Research cluster in March 2012; that study first showed us how well the Pregelix architectural approach can scale.

8. RELATED WORK

The Pregelix system is related to or built upon previous works from three areas.

Parallel data management systems such as Gamma [21], Teradata [8], and GRACE [25] applied partitioned-parallel processing to SQL query processing over two decades ago. The introduction of Google's MapReduce system [20], based on similar principles, led to the recent flurry of work in MapReduce-based data-intensive computing. Systems like Dryad [30], Hyracks [12], and Nephele [11] have successfully made the case for supporting a richer set of data operators beyond map and reduce as well as a richer set of data communication patterns. REX [33] integrated user-defined delta functions into SQL to support arbitrary recursions and built stratified parallel evaluations for recursions. The Stratosphere project also proposed an incremental iteration abstraction [24] using working set management and integrated it with parallel dataflows and job placement. The lessons and experiences from all of these systems provided a solid foundation for the Pregelix system.

Big Graph processing platforms such as Pregel [32], Giraph [4], and Hama [6] have been built to provide vertex-oriented message-passing-based programming abstractions for distributed graph algorithms to run on shared-nothing clusters. Sedge [43] proposed an efficient advanced partitioning scheme to minimize inter-machine communications for Pregel computations. Surfer [17] is a Pregel-like prototype using advanced bandwidth-aware graph partitioning to minimize the network traffic in processing graphs. In seeming contradiction to these favorable results on the efficacy of smart partitioning, literature [29] found that basic hash partitioning works better because of the resulting balanced load and the low partitioning overhead. We seem to be seeing the same with respect to GraphLab, i.e., the pain is not being repaid in performance gain. GPS [36] optimizes Pregel computations by dynamically repartitioning vertices based on message patterns and by splitting high-degree vertices across all worker machines. Giraph++ [38] enhanced Giraph with a richer set of APIs for user-defined partitioning functions so that communication within a single partition can be bypassed. GraphX [40] provides a programming abstraction called Resident Distributed Graphs (RDGs) to simplify graph loading, construction, transformation, and computations, on top of which Pregel can be easily implemented. Different from Pregel, GraphLab [31] provides a vertex-update-based programming abstraction and supports an asynchronous model to increase the level of pipelined parallelism. Trinity [37] is a distributed graph processing engine built on top of a distributed in-memory key-value store to support both online and offline graph processing; it optimizes message-passing in vertex-centric computations for the case where a vertex sends messages to a fixed set of vertices. Our work on Pregelix is largely orthogonal to these systems and their contributions because it looks at Pregel at a lower architectural level, aiming at better out-of-core support, plan flexibility, and software simplicity.

Iterative extensions to MapReduce like HaLoop [15] and PrIter [46] were the first to extend MapReduce with looping constructs. HaLoop hardcodes a sticky scheduling policy (similar to the one adopted here in Pregelix and to the one in Stratosphere [24]) into the Hadoop task scheduler so as to introduce a caching ability for iterative analytics. PrIter uses a key-value storage layer to manage its intermediate MapReduce state, and it also exposes user-defined policies that can prioritize certain data to promote fast algorithmic convergence. However, those extensions still constrain computations to the MapReduce model, while Pregelix explores more flexible scheduling mechanisms, storage options, operators, and several forms of data redistribution (allowed by Hyracks) to optimize a given Pregel algorithm's computation time.

9. CONCLUSIONS

This paper has presented the design, implementation, early use cases, and evaluation of Pregelix, a new dataflow-based Pregel-like system built on top of the Hyracks parallel dataflow engine. Pregelix combines the Pregel API from the systems world with data-parallel query evaluation techniques from the database world in support of Big Graph Analytics. This combination leads to effective and transparent out-of-core support, scalability, and throughput, as well as increased software simplicity and physical flexibility. To the best of our knowledge, Pregelix is the only open source Pregel-like system that scales to out-of-core workloads efficiently, can sustain multi-user workloads, and allows runtime flexibility. This sort of architecture and methodology could be adopted by parallel data warehouse vendors (such as Teradata [8], Pivotal [7], or Vertica [9]) to build Big Graph processing infrastructures on top of their existing query execution engines. Last but not least, we have made several stable releases of the Pregelix system (http://pregelix.ics.uci.edu) in open source form since the year 2012 for use by the Big Data research community, and we invite others to download and try the system. As future work, we plan to automate physical plan selection via a cost-based optimizer (similar to literature [28]) and we plan to integrate Pregelix with AsterixDB [1] to support richer forms of Big Graph Analytics.

Acknowledgements

Pregelix has been supported by a UC Discovery grant, NSF IIS awards 0910989 and 1302698, and NSF CNS awards 1305430, 1351047, and 1059436. Yingyi Bu is supported in part by a Google Fellowship award. The AsterixDB project has enjoyed industrial support from Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, and Yahoo!. We also thank the following people who tried Pregelix at the early stage, reported bugs, hardened the system, and gave us feedback: Jacob Biesinger, Da Yan, Anbang Xu, Nan Zhang, Vishal Patel, Joe Simons, Nicholas Ceglia, Khanh Nguyen, Hongzhi Wang, James Cheng, Chen Li, Xiaohui Xie, and Harry Guoqing Xu. Finally, we thank Raghu Ramakrishnan for discussing this work with us and sponsoring our access to a Yahoo! cluster for scale-testing early versions of the system.

10. REFERENCES

[1] AsterixDB. http://asterixdb.ics.uci.edu.
[2] BTC. http://km.aifb.kit.edu/projects/btc-2009/.
[3] Genomix. https://github.com/uci-cbcl/genomix.
[4] Giraph. http://giraph.apache.org/.
[5] Hadoop/HDFS. http://hadoop.apache.org/.
[6] Hama. http://hama.apache.org/.
[7] Pivotal. http://www.gopivotal.com/products/pivotal-greenplum-database.
[8] Teradata. http://www.teradata.com.
[9] Vertica. http://www.vertica.com.
[10] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.
[11] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.
[12] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.
[13] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.
[14] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.
[15] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.
[16] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.
[17] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, page 3, 2012.
[18] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.
[19] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[20] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[21] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.
[22] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
[23] S. Even. *Graph Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.
[24] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.
[25] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, pages 209–219, 1986.
[26] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.
[27] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
[28] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.
[29] I. Hoque and I. Gupta. LFGraph: Simple and fast distributed graph analytics. In *TRIOS*, 2013.
[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.
[31] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
[32] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
[33] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.
[34] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.
[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
[36] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, page 22, 2013.
[37] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conference*, pages 505–516, 2013.
[38] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.
[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *SoCC*, page 5, 2013.
[40] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on spark. In *GRADES*, page 2, 2013.
[41] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.
[42] D. Yan, J. Cheng, K. Xing, W. Ng, and Y. Bu. Practical pregel algorithms for massive graphs. In *Technique Report, CUHK*, 2013.
[43] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD Conference*, pages 517–528, 2012.
[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computingt. In *NSDI*, 2012.
[45] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research In Genome Research*, 18(5):821–829, 2008.
[46] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: a distributed framework for prioritized iterative computations. In *SoCC*, page 13, 2011.