Asymmetry in Large-Scale Graph Analysis, Explained

Vasiliki Kalavri*[1] Stephan Ewen†[2] Kostas Tzoumas†[3] Vladimir Vlassov*[4]
Volker Markl†[5] Seif Haridi*[6]
*KTH Royal Institute of Technology †Technische Universität Berlin
*[1,4,6]{kalavri,vladv,haridi}@kth.se †[2,3,5]firstname.lastname@tu-berlin.de

ABSTRACT

Iterative computations are in the core of large-scale graph processing. In these applications, a set of parameters is continuously refined, until a fixed point is reached. Such fixed point iterations often exhibit non-uniform computational behavior, where changes propagate with different speeds throughout the parameter set, making them active or inactive during iterations. This asymmetrical behavior can lead to a many redundant computations, if not exploited. Many specialized graph processing systems and APIs exist that run iterative algorithms efficiently exploiting this asymmetry. However, their functionality is sometimes vaguely defined and due to their different programming models and terminology used, it is often challenging to derive equivalence between them.

We describe an optimization framework for iterative graph processing, which utilizes dataset dependencies. We explain several optimization techniques that exploit asymmetrical behavior of graph algorithms. We formally specify the conditions under which, an algorithm can use a certain technique. We also design template execution plans, using a canonical set of dataflow operators and we evaluate them using real-world datasets and applications. Our experiments show that optimized plans can significantly reduce execution time, often by an order of magnitude. Based on our experiments, we identify a trade-off that can be easily captured and could serve as the basis for automatic optimization of large-scale graph-processing applications.

1. INTRODUCTION

Iterations are inevitably in the heart of many graph-parallel algorithms. Commonly, in these algorithms, the task is to iteratively refine the values of the graph vertices, until a termination condition is satisfied. In each iteration, new values of the vertices are computed, using an update function. The algorithm terminates when some convergence criterion is met.

Many iterative graph algorithms expose non-uniform behavior, where changes propagate at different speeds across iterations. Examples include any algorithm that contains some kind of an iterative refinement process. Ideally, one would like to detect this phenomenon and stop the computation early for the inactive parts. This would allow the system to avoid redundant computation and communication. Applying this simple optimization requires detecting inactive vertices and identifying the parts for which computation can halt. However, one must examine how, halting computation for some vertices, could potentially affect the correctness of the computation for the rest of them. In other words, even if inactive parts can be accurately identified, it might not always be possible to halt computation for these parts and still obtain correct results.

To clarify these issues, we use the single source shortest paths (SSSP) algorithm. Consider the graph of Figure 1, where S is the source, the weights on the edges represent distances, i is the iteration counter and the values in the boxes show the distance computed for each vertex after each iteration. In this example, the algorithm is refining the distances of vertices from the source vertex S. In each iteration, a vertex receives new candidate distances from its incoming-neighbors, selects the minimum of these candidates and its current value, adopts this minimum as the new value and then propagates it to its out-going neighbors (in a vertex-centric programming model). For this problem, it is trivial to detect the vertices that have reached convergence; the ones whose value does not change between two consecutive iterations (shown in gray in Figure 1). It is also easy to see that if we halt computation for these vertices (i.e. the vertices do not send or receive any values), the final result will still be correct. For example, in iteration 3, the value of A does not need to be recomputed. Moreover, A does not need to propagate its distance to B or C again. Any future distances they will compute can only be equal or lower than their current values.

Even though the way to apply this optimization to SSSP may seem obvious, it cannot be easily generalized for all similar algorithms. Consider that the given graph is part of a web graph and the task is to iteratively compute PageRank on this graph. In its simplest form, during one iteration of PageRank, a vertex computes its new rank by summing up the weighted ranks of all of its incoming-neighbors. If we try to apply the previously described optimization in this case, and assuming that vertex A converges in iteration 3, then during this iteration, vertex C will only receive the weighted ranks of vertices B and D, sum them up and therefore compute an incorrect value (missing A's contribution).

A large number of highly-specialized systems for large-scale iterative and graph processing have emerged [8, 9, 11], while there also exist general-purpose analysis systems with support for iterations [6, 12, 13]. Specialized systems are usually designed to exploit dataset dependencies, in order to efficiently execute applications and avoid redundant computations. General-purpose systems often do not match the specialized systems in performance, as they typically do not embody sophisticated optimizations for graph processing. Each system requires computations to be expressed using different programming abstractions and it is not always trivial to

GRADES'14, June 22 - 27 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2982-8/14/06 ...$15.00.
http:dx.doi.org10.11452621934.2621940.

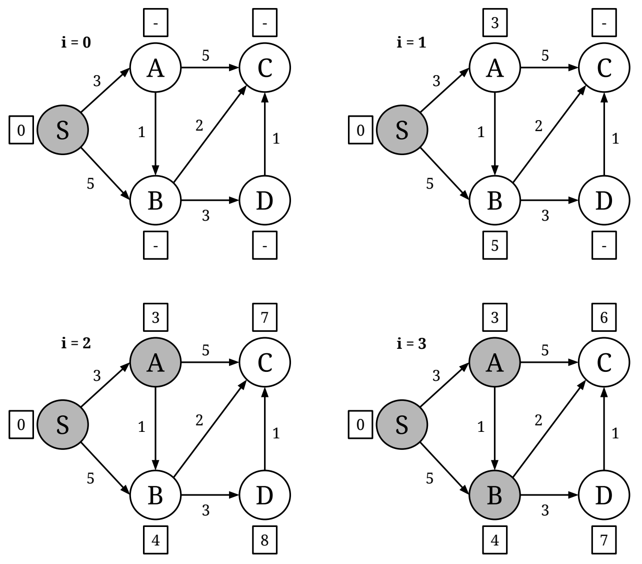


Figure 1: An Example Execution of SSSP

derive a mapping from one model to another. Pregel [9], for example, uses the vertex model and defines that if a vertex does not receive any messages during an iteration, it becomes deactivated and does not execute or produce messages in the subsequent superstep. GraphLab [8] realizes similar behavior with its adaptive execution mechanism. However, it is left to the developer to decide when it is safe to deactivate vertices or halt parts of the computation. This requires the user to understand both models and to carefully verify the correctness of the algorithm.

In this work, we present an overview of general optimizations for graph processing, in the presence of asymmetrical behavior in computations. We study the characteristics of several iterative techniques and we describe what these characteristics mean and how they can be safely exploited, in order to derive optimized algorithms. More importantly, we give the necessary conditions under which, it is safe to apply each of the described optimizations, by exploiting problem-specific properties. We use general-purpose dataflow operators to create template optimized execution plans, which can detect converged parts and avoid redundant computations, while providing functionality equivalent to this of Pregel and GraphLab. We evaluate the optimizations using two characteristic iterative algorithms, Connected Components and PageRank. We present extensive experiments using real-world datasets of varying sizes. We show that optimized algorithms can yield order of magnitude gains compared to the naive execution. Our contributions can serve as the foundation for building a cost-based optimizer that would relieve the programmer from the burden of manually exploiting asymmetry.

The contributions of this paper are the following:

1. A categorization of optimizations for fixed point iterative graph processing, using a common mathematic model.
2. A formal description of the necessary conditions under which the relevant optimizations can be safely applied.
3. A mapping of the optimization techniques to existing graph processing abstractions.
4. An implementation of template optimized execution plans, using general data-flow operators.
5. An experimental evaluation of the optimizations, using a common runtime.

The rest of this paper is organized as follows. In Section A we introduce the notation used throughout this document. In Section 2 we present four different iteration techniques. We describe the conditions, under which, each technique can be used and we make a connection between the described optimizations and existing graph-processing systems. We also describe the implementation of each technique as a template execution plan, while we present our experimental results in Section 3. We discuss related work in Section 4 and conclude in Section 5.

2. GRAPH ITERATION TECHNIQUES

We use common graph notation to explain the different iteration techniques. Let $G\left(V,E\right)$ be an invariant directed graph, where V is the set of vertices and E is the set of directed edges. We also define the following auxiliary problem constructs:

- the *solution set*, containing the values of all vertices in V.
- the *dependency set*, containing the in-neighbors of a vertex v_j and
- the *out-dependency set*, containing the out-neighbors of v_j.

In the SSSP example of Figure 1, the initial solution set would be $S_1 = \{0, na, na, na, na\}$ and the final solution set would be $S_4 = \{0, 3, 4, 6, 7\}$. The dependency sets for each vertex would be $D_S = \emptyset$, $D_A = \{S\}$, $D_B = \{S, A\}$, $D_C = \{A, B, D\}$ and $D_D = \{B\}$ and the out-dependency sets $U_S = \{A, B\}$, $U_A = \{B, C\}$, $U_B = \{C, D\}$, $U_C = \emptyset$ and $U_D = \{C\}$, respectively.

If F is the update function defined over the domain of the values of the vertices of the iteration graph, F is decomposable and has a fixed point, then we can compute the fixed point by executing the following procedure: $S_{i+1} := F(S_i, D)\ until\ S_{i+1} = S_i$.

Next, we describe four graph iteration techniques. We introduce the general bulk technique and then describe three possible optimizations. For each optimization, we give the necessary conditions, under which the optimization is safe. We provide a proof of equivalence with the general-purpose technique in the Appendix.

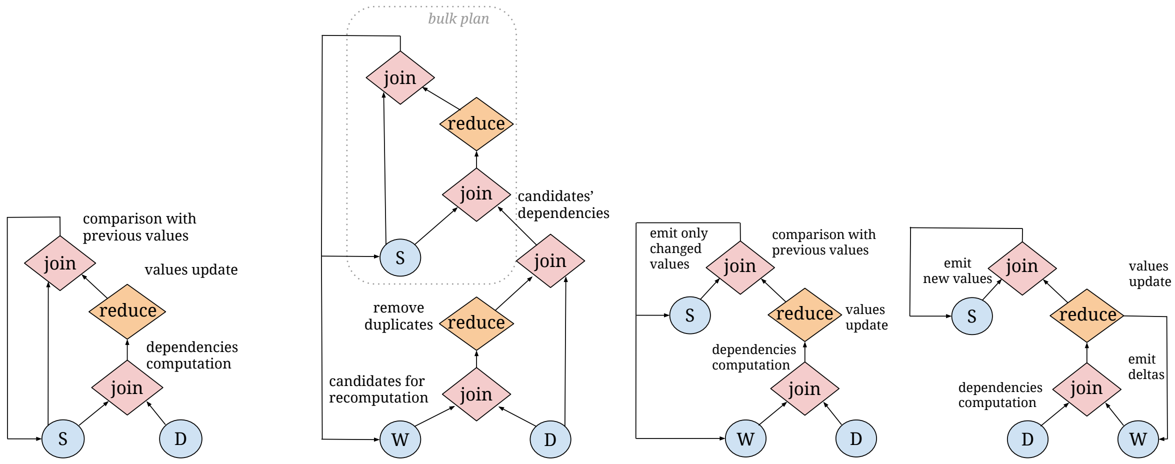
2.1 The Bulk Technique

During a bulk iteration, *all* the elements of the solution set S are recomputed, by applying the update function f to the result of the previous iteration. In the end of each iteration, S is updated with the newly computed values. The algorithm terminates when none of the values of the solution set changes, i.e. the newly computed S in iteration i is identical to the solution set of the previous iteration.

An implementation of the bulk processing model, using common data-flow operators is shown in Figure 2(a). The solution set S contains the vertices of the input graph and the dependency set D contains directed edges (in case of an undirected graph, each edge appears twice, covering both dependencies). In each iteration, the set of vertices is joined with the set of edges to produce the dependencies of each vertex. For each match, a record is emitted, having the target vertex as key. Then, records with the same key are grouped together and the update function is applied. The newly computed values are emitted and then joined with the previous values in order to update the solution set and check the convergence criterion.

2.2 The Dependency Technique

In the bulk case, a value of a vertex, when recomputed, may produce the same result as the one of the previous iteration. This may



(a) Bulk plan (b) Dependency plan (c) Incremental plan (d) Delta plan

Figure 2: Graph processing techniques as data flow plans.

Iteration Technique	Equivalent to Bulk?	Vertex Activation	Vertex Update
Bulk	n/a	always	using values of all in-neighbors
Dependency	always	if any in-neighbor is updated	using values of all in-neighbors
Incremental	f idempotent and weakly monotonic	if any in-neighbor is updated	using values of changed in-neighbors
Delta	f is linear over composition operator	if any in-neighbor is updated	using values of changed in-neighbors

Table 1: Iteration Techniques Equivalence

happen because (a) either none of the values in the dependency set of the vertex has changed since the previous iteration or (b) applying the update function to the changed values happens to return an identical result. Ideally, we would like to only recompute the values of the vertices that are *guaranteed* to change value during an iteration. Instead, we can exploit the dependency set to safely select the vertices that are *likely* to change value in the next iteration.

We introduce two auxiliary sets, the *workset* W and the *candidate set* Z. In each iteration, W stores the vertices which have changed value since the last iteration and Z stores the candidate vertices for recomputation. In other words, Z contains the vertices whose *at least one* in-neighbor has changed value during the last iteration. Z is essentially an overestimation of the ideal set of vertices that are guaranteed to require recomputation.

The intuition behind this algorithm is that if a vertex of the iteration graph changes value, then, all its out-neighbors are likely to be affected by this change. On the other hand, if none of the dependencies of a vertex changes value, it is safe to exclude this vertex from the next computation, since recomputing the update function on the same arguments, would return an identical result.

An implementation of the dependency processing model is shown in 2(b). The workset W is isomorphic to the solution set S and contains only the records corresponding to the vertices that changed value during the previous iteration. W is joined with the dependency set to generate candidates, emitting a record with the target vertex id as the key for each match. The candidate vertices for recomputation are grouped to remove duplicates and joined with the dependency set on target vertex id, producing a subset of the dependency set. The resulting set serves as input to a subplan equivalent to the previously described bulk model, which only recomputes the new values of the vertices, whose at least one input changed during the previous iteration. The last join operator, only emits records containing vertices that changed values back to the workset.

2.3 The Incremental Technique

In some cases, the dependency technique can be further optimized. Specifically, if the function f_j is of the form $f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n$, where $t_1, t_2, \cdots t_n$ represent independent contributions and f_j is *distributive* over the combination operator $\sqcup$, then we can optimize by only computing f_j on the changed values of the dependency set in each iteration and then, combine the result with the previous value. For example, if t is the identity function and the combination operator is $minimum$, then $f_j = min(t(D_j)) = t(min(D_j))$. In the graph of Figure 1, the value of node B depends on the values of nodes S and A, thus, $D_B = \{S, A\}$. Then, $f_B = min(t(value(S)), t(value(A))) = t(min(value(S), value(A))) = min(value(S), value(A))$. In the Appendix, we prove that, if f is also *idempotent* and *weakly monotonic*, then the combination can be reduced to applying f_j to the previous value and the partial result. Returning to the SSSP example, $minimum$ is also idempotent ($min(a, a) = a$) and also weakly monotonic, since, for $a \leq a'$ and $b \leq b'$, $min(a, b) \leq min(a', b')$.

The incremental technique uses the introduced above workset W, which, in every iteration, contains the elements of S that changed value since the last iteration and the candidates set Z, which contains the candidate elements for recomputation. Its implementation using common data-flow operators is shown in Figure 2(c). This execution plan takes W as input, which stores only the vertices with changed values. First, the workset is joined with the dependency set to generate the candidate set Z. The result is grouped by key and the update function is applied to each group. After the new values have been computed, only the parameters whose value has changed are emitted into the workset.

2.4 The Delta Technique

Ideally, for each change δx in the input, we would like to have an efficient function δF, such that: $F(x \oplus \delta x) = F(x) \oplus \delta F(x, \delta x)$ where $\oplus$ is a binary composition operator. In this ideal scenario, we could propagate only the differences of values, or *deltas*, from each iteration to the next one. That would potentially decrease the communication costs and make the execution more efficient. However, there are two major factors one has to consider. First, it might not always be the case that computing $\delta F(x, \delta x)$ is more efficient than simply computing $F(x \oplus \delta x)$. Moreover, even if we are able to find an efficient δF, combining its result with $F(x)$ could still prove to be a costly operation. In the special case where the update function f is *linear* over the composition operator $\oplus$, then $F(x \oplus \delta x) = F(x) \oplus F(\delta x)$, in which case we can use the same function f in the place of δf.

For example, if $f = sum(D)$, this optimization is applicable. Let us assume that $D^i = \{a, b\}$ and $D^{i+1} = \{a', b\}$, where $a' = a + \delta a$. Then, $f^{i+1} = sum(a', b) \Rightarrow f^{i+1} = sum(a + \delta a, b) \Rightarrow f^{i+1} = sum(a, b, \delta a) = sum(f^i + \delta a)$.

A data-flow execution plan implementing the delta technique is shown in Figure 2(d). In this plan, only the differences of values are propagated to the workset.

Table 1 summarizes the equivalence among the different techniques and the conditions for safely applying each optimization.

2.5 Iteration Techniques in Iterative and Graph Processing Systems

In the Pregel [9] model, an iteration corresponds to one superstep. The vertex-centric Pregel model naturally translates to the incremental iteration technique. The vertices receive messages from neighboring vertices and compute their new value using those messages only. The candidates set Z can be seen as maintaining the subset of the active vertices for the next superstep. The delta iteration technique can be easily expressed using the vertex-centric-model, if vertices produce deltas as messages for their neighbors. To emulate a bulk iteration in the Pregel model, vertices simply need to transfer their state to all their neighbors, in every iteration. Vertices would remain active and not vote to halt, even if they do not have an updated state. Finally, emulating the dependency iteration in Pregel is not that trivial, since vertices in Pregel can only send messages to their out-neighbors. However, in the dependency technique, if a vertex is candidate for recomputation, it needs to *activate* all its in-coming vertices, therefore, it needs to send them a message. A way around this would be to add a pre-processing step, where all vertices send their ids to their out-neighbors. Then, when a vertex receives messages from the in-neighbors, it can use the ids to create auxiliary out-going edges to them. The computation could then proceed by using a three-step superstep as one dependency iteration: during the first step, vertices with changed values produce messages for their out-neighbors. During the second step, vertices that receive at least one message are candidates for recomputation and produce messages for all their in-neighbors, while the rest of the vertices become inactive. In the third step, the candidates for recomputation receive messages from all their in-neighbors and update their value.

GraphLab's [8] programming abstraction consists of the data graph, an update function and the sync operation. The data graph structure is static, similar to what we assume for the dependency set. GraphLab introduces the concept of the scope of a vertex, which is explicitly declared and refers to the set of values of a vertex and its neighbors. This scope corresponds to the dependency set in the bulk and dependency techniques and to the intersection of the dependency set of a vertex and the workset, in the incremental and delta iterations. Therefore, all four algorithms can be implemented by GraphLab, by computing the appropriate scopes.

PowerGraph [7] is a graph processing system for computation on natural graphs. It introduces the Gather-Apply-Scatter (GAS) abstraction, which splits a program into these three phases. During the gather phase, a vertex collects information from its neighborhood, which then uses during the apply phase to update its value. During the scatter phase, the newly computed values are used to update the state of adjacent vertices. The GAS abstraction can be used to implement both the Bulk and the Incremental iteration plans, while the Delta plan is equivalent to PowerGraph's delta caching mechanism. The model does not intuitively support the dependency technique. However, it can be implemented in a similar way to the three-step superstep described for Pregel.

System	Bulk	Dependency	Incremental	Delta
Pregel	**	***	*	**
GraphLab	*	*	**	**
GraphX	**	***	*	**
Powergraph	*	***	*	*
Stratosphere	*	**	*	**

Table 2: Iteration Techniques in Graph and Iterative systems. *: provided by default, **: can be easily implemented, *: possible, but non-intuitive**

GraphX [12] is a graph processing library built on top of Spark [13], in order to efficiently support graph construction and transformation, as well as graph parallel computations. The programming model of GraphX is equivalent to that of Pregel and PowerGraph.

Stratosphere [2, 3] supports flexible plans in the form of a Directed Acyclic Graph (DAG) of operators. Iterations are implemented in Stratosphere as composite operators, which encapsulate the step function and the termination criterion. The implementation of the Bulk and the Incremental algorithms are described in [6]. Nevertheless, all of the algorithms described above can be easily implemented in Stratosphere.

Table 2 summarizes the support of each technique in popular graph processing and iterative systems. Most of the existing systems implement the bulk technique by default and special implementations of operators to support the delta optimization. These models assume that the update function has the required characteristics, or that it can be easily re-written, in order to fulfill the required conditions and, therefore, usually do not expose the implementation of an equivalent to the more general dependency technique. Indeed, it is usually trivial to derive an incremental or delta version of a typical aggregate update function. Apart from the cases when an incremental/delta version of the update function cannot be easily derived, the dependency technique can prove to be beneficial in cases when the properties of the update function are not known to the user, for example, if the function used belongs to an external library and the user has no access to its source code.

3. EXPERIMENTS

In this Section, we evaluate optimization strategies for graph processing, by examining two popular iterative algorithms. We implement the applications in Stratosphere [2, 3], an open-source general-purpose framework for large-scale data analysis, which has operators for common processing tasks, such as mapping, group and join. In [6], it is shown that Stratosphere iterations are comparable in performance with Spark and Giraph, even without using the optimizations discussed in this paper.

Our setup consists of an OpenStack cluster, using 9 virtual machines, each having 8 virtual CPUs, 16 GB of memory and 170 GB of disk space. Nodes run Linux Ubuntu 12.04.2 LTS OS.

We evaluate the performance of two iterative algorithms, Connected Components and PageRank. The update function of the

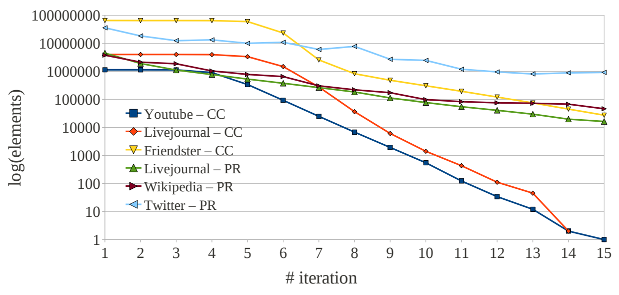


Figure 3: Updated Elements per Iteration. CC: Connected Components, PR: PageRank

Connected Components algorithm (minimum) satisfies the conditions of the incremental technique. Therefore, we implement this application using the Bulk, Incremental and Dependency plans. We execute the Connected Components plans using datasets from [1]. The update function of PageRank (summation of partial ranks) satisfies the conditions of the delta technique. Therefore, we implement this application using the Bulk, Delta and Dependency plans. Initial deltas can be derived from the difference between the uniform initial rank and the in-degree proportional rank. We execute the PageRank plans using datasets from SNAP [1] and the Laboratory for Web Algorithmics respectively [4, 5].

3.1 Results

Figure 3 shows the number of elements that actually change value in each iteration, for both the Connected Components and PageRank experiments. If this behavior is not accounted for, a lot of redundant computations will be performed. We also observe that the intensity of the phenomenon differs among datasets and depends on the dependency graph properties and the algorithm.

In Figures 4 and 5, we present execution time measurements for the Connected Components and PageRank algorithms, respectively. In each figure, we have plotted the execution time per iteration for the Bulk, Dependency and Incremental or Delta plan implementations. As expected, the time for all Bulk iterations is quite stable throughout the execution, for all the cases examined. Regarding the Dependency plan, we observe that in the first few iterations, it is consistently less efficient than the Bulk plan. This is due to the fact that the Dependency plan first needs to identify the candidate elements for recomputation and retrieve their dependencies. This pre-processing step imposes an overhead compared to the Bulk execution. When the amount of elements in the workset is close to the total amount of elements in the solution set, the overhead of the pre-processing is larger than the time saved by updating less elements. In the case of the Connected Components, the Dependency plan outperforms the Bulk plan in later iterations, for both the Livejournal and Friendster datasets. For the Livejournal dataset (Figure 4(a)), the execution time of the Dependency plan drops significantly after iteration 7. As seen in Figure 3, it is this iteration that the elements in the workset also greatly decrease. Regarding the Friendster dataset (Figure 4(b)), the execution time of the Dependency plan outperforms the Bulk plan after iteration 8, but its cost remains more or less stable until convergence, due to the much slower decrease rate of the workset elements, as seen in Figure 3. In the case of PageRank, we observe similar behavior for the Livejournal and Wikipedia datasets in Figures 5(a) and 5(b). The Dependency plan cost keeps decreasing as the workset is shrinking across iterations. Regarding the Twitter dataset, Figure 5(c) shows that the overhead of the pre-processing step remains dominant in the Dependency execution, during all iterations. This is in accordance to Figure 3, where we observe a much smaller decline in the workset size of the Twitter dataset, compared with the other datasets.

Unsurprisingly, the Incremental and Delta plans match the Bulk plan performance during the first iterations, while they significantly outperform it as the workset shrinks. In all cases, the optimized plans outperform the Bulk plan by an order of magnitude or more, as iterations proceed. Finally, we observe that the Delta plan always performs better than both the Bulk and the Dependency plans and its cost continuously decreases.

Our experiments show that the Incremental and Delta plans save a lot of redundant computations and should always be preferred over Bulk plans. However, these plans can be used only when the update function of the algorithm satisfies the conditions described in Section 2 for the incremental and the delta techniques respectively. What is more interesting to examine is when and how the more general Dependency plan can be used to speed up total execution time. Our results show that there is a trade-off that depends on the size of the workset. We intend to build a cost model that will be able to capture the overhead of the Dependency plan over the Bulk plan. We plan to use this cost model and the results of our analysis to build a cost-based optimizer to choose the most efficient iteration plan, at runtime.

4. RELATED WORK

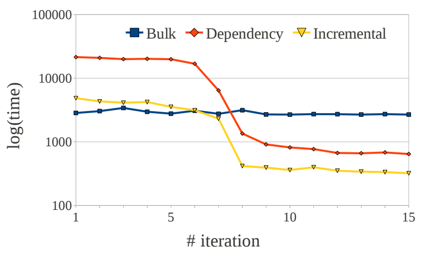
To the best of our knowledge, no directly related work exists that categorizes and formalizes optimizations for large-scale graph processing. GraphX [12] is the work closest to ours. Like us, the authors realize that graph computations can be expressed using common relational operators, including joins and aggregations. Regarding optimizations, the system supports incremental computation, by maintaining a replicated vertex view, in memory. Our proposed execution plans are more general and do not rely on maintaining views, in order to implement incrementalization.

Delta and Incremental optimizations have been used in several other systems as well. REX [10] is a system for recursive, delta-based data-centric computation, which uses user-defined annotations to support incremental updates. It allows explicit creation of custom delta operations and lets nodes maintain unchanged state and only compute and propagate deltas. The system optimizer discovers incrementalization possibilities during plan optimization, while the user can also manually add delta functions to wrap operators, candidates for incrementalization. Naiad [11] is a stream processing framework which supports computation on arbitrarily nested fixed points. Naiad assumes a partial order, keeps a timestamped full-version history and responds to additions and subtractions of records by maintaining state.

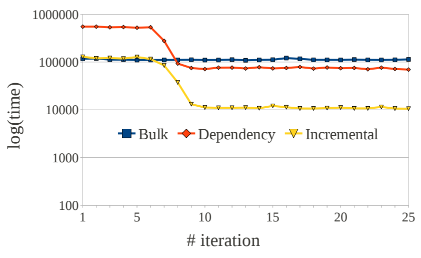
5. CONCLUSIONS AND FUTURE WORK

In this work, we present a taxonomy of optimizations for iterative fixpoint algorithms. We describe ways to exploit asymmetrical behavior to implement optimized execution plans. We offer proof of equivalence between different approaches and we provide a mapping to existing iterative and graph processing programming models. We implement template execution plans, using common dataflow operators and we present experimental evaluation, using a common runtime. Our results demonstrate order of magnitude gains in execution time, when the optimized plans are used.

In the future, we plan to design a cost model to accurately predict the cost of subsequent iterations. Then, using information about the changed elements in the current iteration, a cost-based optimizer could be used to choose the most efficient execution plan, at run-

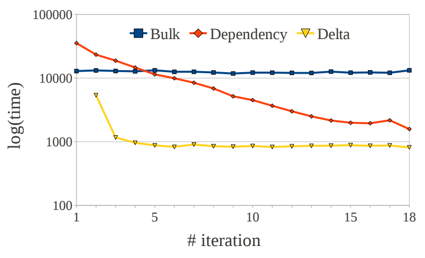


(a) Livejournal

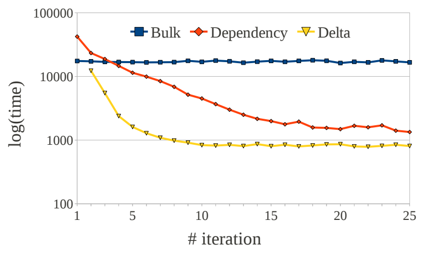


(b) Friendster

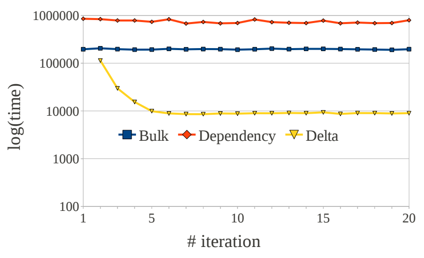
Figure 4: Connected Components Execution Time per Iteration



(a) Livejournal



(b) Wikipedia



(c) Twitter

Figure 5: PageRank Execution Time per Iteration

time. We also intend to extend our proposed iteration plans to more general iterative algorithms. We plan to implement a set of iterative applications and compare performance with other iterative and graph processing systems.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043. Stratosphere is supported in part by the DFG (German Research Council), the FP7 Program of the European Union, and EIT ICTLabs.

6. REFERENCES

[1] SNAP: Stanford Network Analysis Platform. http://snap.stanford.edu/index.html. [Online; Last accessed January 2014].

[2] Stratosphere. http://getstratosphere.org. [Online; Last accessed January 2014].

[3] 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 *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, 2010.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, 2011.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004.

[6] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 2012.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[9] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[10] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: Recursive, delta-based data-centric computation. *Proc. VLDB Endow.*, 5(11):1280–1291, 2012.

[11] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[12] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, New York, NY, USA, 2013. ACM.

[13] 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 computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

APPENDIX

A. NOTATION

An instance of a fixed point iteration problem consists of the following constructs:

- A *solution set* $S = \{x_1, x_2, ..., x_n\}$, which contains a finite number of variables x_j, for which we want to reach convergence. The problem is considered solved when all variables converge to their final values, so that none of the values in the set is affected by a subsequent iteration.
- An *update function* $F : domain(x_j) \rightarrow domain(x_j)$, decomposable to partial functions $f_1, f_2, ...f_n$, for computing each $x_j \in S$, where $domain(x_j)$ is the domain of values in which x_j are defined and $j \in [1, n]$.
- A *dependency set* $D = \{D_1, D_2, ..., D_n\}$, where $D_j \subseteq S$, describes the dependency relation of element x_j on other elements of S and $j \in [1, n]$. D_j therefore contains the elements of S on which the computation of x_j's value depends. In other words, D_j contains the elements of S which appear on the right hand side of the function f_j.

From the dependency set D, we derive the auxiliary *out-dependency* set, $U = \{U_1, U_2, ..., U_n\}$, where $U_j \subseteq S$. U contains the elements of S whose values depend on x_j and is formally defined as $U_j = \{x_k \mid x_k \in S \, and \, x_j \in D_k\}$, $j, k \in [1, n]$. In the rest of this paper we assume familiarity with standard set notation. Indices k, j are used to refer to elements of S and its subsets, while index i is used to denote an instance of a construct during the i-th iteration.

B. PROOFS

PROPOSITION 1. *The Dependency Algorithm is equivalent to the Bulk Algorithm.*

PROOF. We prove this statement using the method of contradiction. Let us assume that the two algorithms are not equivalent. Then, there exists an initial input set S_0 and a function f for which the two algorithms converge to different solution sets. Let us assume that the algorithms give identical partial solution sets until iteration i, but the results diverge in iteration $i + 1$. If S_b^{i+1} is the partial solution set produced by the execution of the Bulk Algorithm and S_w^{i+1} is the partial solution set produced by the execution of the Dependency Algorithm after iteration $i+1$, there should exist at least one element that is different in the two sets.

Since W is a subset of S, that would mean that the Dependency Algorithm failed in identifying all the vertices that required recomputation during iteration i, i.e. there exist $x_{j,b}^i \in S_b^i$, $x_{j,b}^{i+1} \in S_b^{i+1}$, $x_{j,z}^i \in S_w^i$, $x_{j,w}^{i+1} \in S_w^{i+1}$, such that

$$x_{j,b}^i \neq x_{j,b}^{i+1} \tag{1}$$

and

$$x_{j,w}^i = x_{j,w}^{i+1} \tag{2}$$

From the relations 1 and (4.1.1) we can derive the following relation:

$$f_j(D_j^{i-1}) \neq f_j(D_j^i)$$

From the relations 2 and (4.2.1) we can derive the following relation:

$$f_j(D_j^{i-1}) = f_j(D_j^i)$$

and we have therefore arrived at a contradiction. □

PROPOSITION 2. *If f_j is of the form $f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n$, where $t_1, t_2, \cdots t_n$ represent independent contributions to the value of f_j, i.e. f_j is distributive over the combination operator $\sqcup$ and f_j is also idempotent and weakly monotonic, then the Incremental Algorithm is equivalent to the Bulk Algorithm.*

PROOF. Let x_j^i be the value of an element of S in iteration i. Since f_j is distributive over $\sqcup$ and idempotent then $x_j^i = f_j(t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n) = f_j(T \sqcup t_n)$, where $T = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_{n-1}$. Let us assume that during iteration i, only t_n changed value and therefore

$$x_j^{i+1} = f_j(T \sqcup t'_n) \tag{3}$$

Since f_j is idempotent,

$$f_j(t_n, t_n) = t_n$$

and

$$f_j(t'_n, t'_n) = t'_n$$

Let us also assume that f_j is weakly increasing (the case of decreasing is analogous). Then, we have the following two cases:

- Case 1: $t'_n < t_n : f_j(T \sqcup t'_n) \leq f_j(T \sqcup t_n) \Rightarrow$
$$f_j(T \sqcup t'_n) \leq f_j(T \sqcup t_n \sqcup t'_n) \tag{4}$$
- Case 2: $t'_n > t_n : f_j(T \sqcup t'_n) \geq f_j(T \sqcup t_n) \Rightarrow$
$$f_j(T \sqcup t'_n) \geq f_j(T \sqcup t_n \sqcup t'_n) \tag{5}$$

From equations 4 and 5 we conclude that

$$f_j(T \sqcup t'_n) = f_j(T \sqcup t_n \sqcup t'_n)$$

Consequently, equation 3 becomes: $x_j^{i+1} = f_j(T \sqcup t_n \sqcup t'_n) \Rightarrow x_j^{i+1} = f_j(x_j^i \sqcup t'_n)$. □