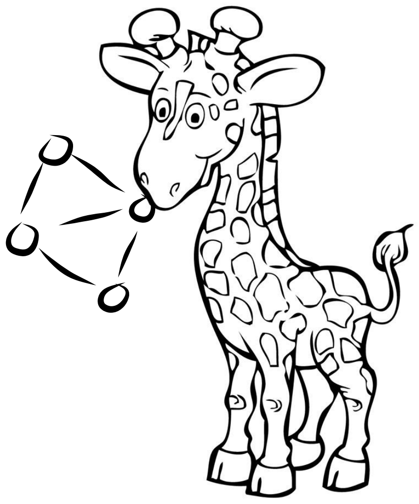
OPTIMIZING SYNCHRONOUS ONLINE COMPUTATION OF LARGE GRAPHS

ALEXANDRE JORGE FONSECA



A thesis submitted in partial fulfillment for the degree of
European Master in Distributed Computing

Developed during internship at Telefónica

Universitat Politècnica de Catalunya

January 2014 - June 2014 – version 1.0

Alexandre Jorge Fonseca: *Optimizing synchronous online computation of large graphs,* European Master in Distributed Computing, © January 2014 - June 2014

SUPERVISORS:
Dionysios Logothetis (Telefónica)
Georgos Siganos (Telefónica)
David Carrera (UPC)

LOCATION:
Barcelona, Spain

TIME FRAME:
January 2014 - June 2014

Dedicated to my mother.

ABSTRACT

The area of big data analytics has witnessed tremendous growth during the last few years. The increased spread and quantity of raw data sources in conjunction with the continuous upgrading of storage mechanisms have led organizations to handle never before seen amounts of data. Much of this data is usually represented as graphs, describing continuously evolving relations such as a social network. As the need for extracting actionable intelligence from this data grows more dire, it is being submitted to increasingly complex algorithms. Traditional relational storages are not optimized for the storage and analysis of graphs and are, therefore, unable to cope with the execution of these novel types of analytics over large scale datasets. This has motivated the development of several large scale distributed graph processing systems such as Pregel and Graphlab.

Pregel-based systems employ a synchronous execution model that, being more familiar to developers, results in greater ease of programming and debugging and allows for good scalability. However, to maintain synchronicity, coordination between all nodes of the cluster is needed at the end of each iteration. In the presence of straggler nodes or network jitter, this global coordination enforces the whole cluster to operate at the speed of the slowest nodes. Graphlab-based systems attempt to mitigate this inefficiency by employing a completely asynchronous execution model where node coordination is greatly relaxed. Only, in doing so, these systems lose the simplicity of the synchronous programming model and start requiring complex agreement protocols.

In this thesis, an attempt is made to reconcile the two execution models by reducing the impact of inter-node coordination in the efficiency of synchronous executions. This is achieved through three different mechanisms: the first, event pipelining, enables the concurrent execution of analytics over different mutations of an evolving graph, using time otherwise spent coordinating to do useful computation; the other two, partition-level and vertex-level local synchronization, track dependencies between nodes, allowing execution to continue even before the coordination phase for the previous step finishes. All 3 mechanisms were evaluated using real-world graphs and applications and have shown to be able to provide considerable improvements to the latency of graph analytics.

If knowledge can create problems, it is not through ignorance that we can solve them.

— Isaac Asimov

ACKNOWLEDGMENTS

First and foremost, I would like to thank my great supervisors at Telefónica — Dionysios Logothetis and Georgos Siganos — for their invaluable guidance and support over the course of this semester as well as for giving me the chance to work on a project which I genuinely enjoyed. Gratitude is also extended to all my other colleagues at Telefónica whose cheerful attitude and interesting exchanges made it an absolute pleasure to come to "work" everyday, specially Mário Almeida, the fellow Portuguese whose antics and rants provided welcome distractions in periods of stress; Yan Grunenberger, the Twitter-parsing expert, for all our discussions and crazy project ideas; Giorgos Dimopoulos, the hardworking Greek, with whom I had many a lunch and Illias Leontiadis, for manning the other side of the screen-wall. I would further like to thank my academic supervisor, David Carrera, for his advice and availability.

A big thank you is also owed to all my EMDC classmates who, during these 2 years, became, in a sense, my overseas family and with whom I shared memorable moments which I will surely treasure for the rest of my life. Furthermore, I would also like to thank Johan Montelius, Leandro Navarro and the rest of the team responsible for setting up and maintaining the EMDC programme, giving me a chance to be part of it and ensuring that the experience was as smooth, entertaining and educational as possible.

Of course my stay in Barcelona wouldn't have been half as sweet without the presence and support of my lovely girlfriend Daniela whose patience and charm are boundless. I would also like to express my gratitude to Carlos Letra, an old family friend, and his girlfriend Maria João, who made the initial transition to Barcelona that much smoother and with whom I enjoyed many an interesting weekend.

Last but never least, I would like to thank my mother whose support and counsel are greatly treasured and without whom this would not have been possible.

Thank you all!

Barcelona, Spain, June 2014

Alexandre Jorge Fonseca

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 GRAPH ANALYTICS

Organizations are increasingly relying on data mining to boost the efficiency and efficacy of their processes. By collecting raw data originated in sensors, activity logs and other data sources and then submitting them to analysis, they are able to acquire actionable intelligence which may be used to identify hidden information of interest to their business or activity: stores might use client, product and purchase data to identify patterns in the shopping habits of a demography of their clientele and tailor promotions or suggestions to incentivize spending; social networks can analyze users, their posts and their acquaintances to suggest new friendships; advertising agencies may analyze user's browsing routines to offer more targeted advertising.

This data composed of different entities interconnected through relationships is naturally modeled through a graph structure where vertices represent the entities of a domain while edges represent relationships between entities. Figure 1 illustrates such a graph for an online shop like Amazon.

The analysis of individual disconnected entities might be able to provide some of this knowledge. However, more intelligence is often hidden in the intrinsic relationships between the entities of a domain. Looking at the data of a single client, for instance, might provide some insight into what its preferences might be. Examining the relationships between a client, products of a store (such as its purchase or review history) and other clients, can generate insight into what its preferences actually are and, thus, help detect purchase social trends and provide richer recommendations.

Traditionally, this kind of data has been stored in centralized relational databases where different types of entities and relationships are stored in separate tables. Figure 2 showcases the equivalent relational model for the graph in Figure 1. Navigation and operations over these tables are done through the use of relational algebra operators abstracted over high-level languages such as SQL. Due to the generic nature of these operators, their use for graph processing is not only unnatural (requiring unnecessarily verbose queries) but also poorly optimized.

The reason for this is that graph-structured data and graph analytics have several characteristics that, when taken into account, distinguish them from other types of data and analytics:

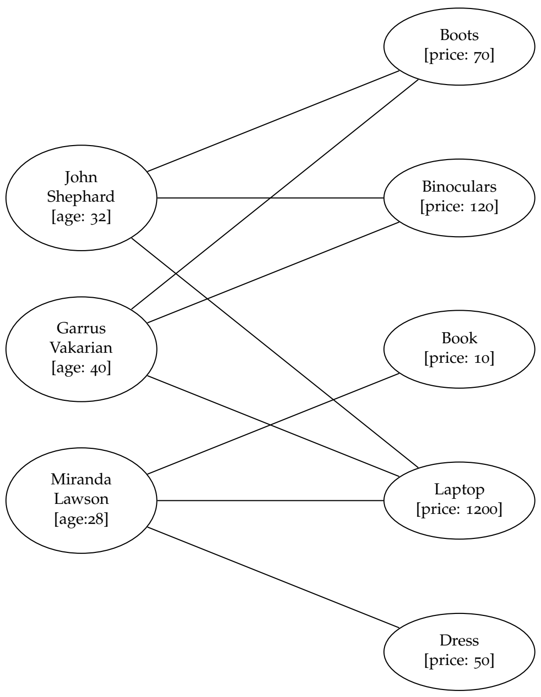


Figure 1: Example graph for an online shop. Ellipses correspond to vertices of the graph and represent domain entities: on the left, clients; on the right, products. Lines connecting vertices correspond to edges and represent relationship between domain entities. In this graph the represented relationship is one of purchase.

Clients			
ID	Name	Age	...
1	John Shephard	32	...
2	Garrus Vakarian	40	...
3	Miranda Lawson	28	...
	...		

Products			
ID	Name	Price	...
1	Boots	70	...
2	Binoculars	120	...
3	Laptop	1200	...
	...		

Purchases			
ClientID	ProductID	Date	...
1	1	2014/01/23	...
1	2	2014/01/27	...
1	3	2014/01/28	...
2	1	2014/01/05	...
	...		

Figure 2: Example relational database for an online shop. Each entity type is stored in a separate table with entities being represented by rows and relationships mapped through the use of a many-to-many relationship table, primary keys (*ID*) and foreign keys (*ClientID* and *ProductID*)

- Graphs can grow to very large sizes. Facebook, for instance, now manages graphs with more than 1 billion vertices and hundreds of billions of edges [9].
- Graph analytics are putting increasing emphasis on the processing of inter-entity relationship chains, requiring iterative traversal over several levels of entity connections.
- Graphs are highly dynamic with new edges and vertices constantly being incorporated and removed to/from them. As an example, on the 1st of December 2013, Amazon achieved a record-breaking number of 426 Prime purchases per second [6], with each such purchase representing an extra edge on the purchase graph exemplified in Figure 1.
- Graph analytics need to be kept up to date with the state of the domain to remain accurate and effective.

Therefore, in recent years, considerable effort has been dedicated to the creation of novel distributed and scalable graph processing architectures, of which Pregel [17] and Graphlab [16] are two of the most significant examples. These systems provide specialized optimized APIs for batch graph processing, allowing the execution of graph analytics that can not only be expressed in a more natural and succinct manner but can also execute in a fraction of the time. In addition, they are designed from the ground up with scalability in mind and perform admirably well in large-size clusters.

1.2 PROBLEM

Pregel (and derived systems) employ a synchronous execution mechanism based on the Bulk Synchronous Parallel (BSP) model [31]. This model, which is described in detail in Section 2.1 divides application execution in a series of supersteps, with each superstep encapsulating computation and communication phases. Execution progress and eventual termination is achieved by progressing through consecutive supersteps until such a time as there is no more work to be done. BSP offers great programming simplicity and should be familiar to most developers due to its synchronous nature. In addition, executions progress deterministically, easily allowing one to determine the state of the computation at any time and, thus, facilitating its debugging and understanding. However, this comes at a price: to enforce synchronous execution, the advancement through supersteps requires all nodes of the cluster to coordinate at the end of each superstep. This is needed to ensure that all required data for the execution of the next superstep is available at each node. This type of coordination involving the entire system is known as global synchronization and has serious inefficiencies in the presence of straggler

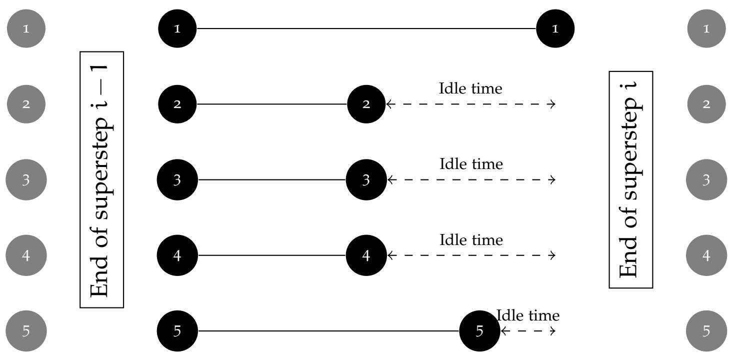


Figure 3: Example superstep execution with straggler nodes. Solid lines represent periods in which nodes execute both the computation and communication phases. Dashed lines represent periods in which nodes are idle. The total time necessary for executing the superstep is equal to the total time necessary by the straggler (node 1).

nodes. These nodes, due to factors such as application execution imbalances or hardware/software problems, require more computation time than others in the same cluster. Because global synchronization requires coordination among the entire cluster, the global execution speed is bottlenecked by the stragglers as shown in Figure 3. In this situation, the non-stragglers cannot advance and computational resources are wasted in idle time, reducing the efficiency and latency of these systems.

Graphlab and similar systems attempt to tackle this issue by using an asynchronous execution model which greatly relaxes the need for coordination between nodes and supersteps. With this model, each node executes at its own pace with whatever information is available to it at the time. While this effectively eliminates the idle times observed with Pregel-based systems, the asynchronous nature of the execution makes algorithm design, programming and debugging considerably harder: without clearly defined computation phases, identifying the current state of the system becomes a complex operation and the non-deterministic progress of the execution makes it very difficult to reproduce experiments. Determining the end of a computation or any other kind of agreement between the various nodes now also requires the use of more sophisticated protocols which result in extra computation time and message exchange.

Given the issues identified with the two described approaches, it is natural to consider whether another solution exists that would conciliate the programming simplicity of synchronous executions with the mitigation of coordination idle time obtained with the asynchronous model.

1.3 CONTRIBUTIONS

This thesis explores mechanisms that aim to improve the efficiency of graph analytics in systems based on the Pregel model. This is done by taking advantage of the idle times observed with global synchronization under the presence of computational skew or network jitter. Ultimately, these mechanisms allow the improvement of both latency and throughput in the execution of graph analytics on these systems.

These mechanisms were designed, implemented and evaluated in the context of RTGiraph, a fork of Giraph (the most popular Pregel open source implementation) designed for online incremental graph computation. RTGiraph employs concepts of incremental computation to memoize intermediate results of previous executions of graph analytics on an underlying graph. These intermediate results are then used to considerably speedup the computation of analytics over mutations (also known as events) of the initial graph. However, despite the focus in RTGiraph, the mechanisms here described should be adaptable to other graph processing systems.

During the course of this thesis work, significant changes were made to the existing codebase of RTGiraph (containing several tens of thousands of lines of code), touching upon the majority of the components of this distributed system, from I/O management to execution scheduling and inter-node coordination. Performed evaluations required the setup and maintenance of large EC2 clusters as well as the usage of real-world graphs and applications.

The next few sections provide a brief overview of the 3 idleness-exploiting mechanisms that were considered in this thesis.

Event pipelining

Graphs are continuously being updated with new mutations. RT-Giraph is able to handle the updating of graph analytics over these mutation in a very efficient manner by reusing results from previous executions. However, depending on the mutation rate, it might still happen that new events arrive before a previous one might have had the chance to finish processing.

Under normal conditions, the execution of those new events would have to be delayed until the currently executing one finishes processing. Only when this happens can the following event proceed. Meanwhile, idle times occurring during the execution of a single event result in completely wasted computational resources.

However, by exploiting the data dependencies described in detail in Section 4.1, the concurrent execution of consecutive events over the same underlying graph is possible. The event pipelining mechanism accomplishes this by using a conceptual pipeline in which the processing of each superstep is seen as a different stage and different events can be at different execution stages inside this pipeline. The scheduling of events in the pipeline is done in such a way so as to guarantee the serializability of parallel executions, thus ensuring that results are equivalent to those that would otherwise be obtained if the considered events had been executed sequentially, one at a time. Having several events running simultaneously allows one to fill the idle times of a computation of an event with processing for the other computations.

Evaluation of this mechanism has shown that it can have a great impact in processing performance with improvements of up to 177% being observed.

Partition-level local synchronization

Instead of taking advantage of idle times to perform useful computations ahead of time like what is done with the previous mechanism, partition-level local synchronization attempts to completely remove idle time in those cases where one already has all the information needed to proceed with execution, even if some of the other nodes are still executing the current superstep.

This is accomplished by replacing the per-node global synchronization barriers used by the Pregel model at the end of every superstep with per-graph-partition local synchronization barriers. These barriers, in conjunction with a dependency identification and tracking mechanism, are able to advance the computation of certain graph partitions to the next superstep whenever all of its dependencies have finished the current one (independently of whether or not other partitions on which it does not depend have also finished).

Experiments with this mechanism have shown that it is particularly effective when the partition meta-graph (the graph representing partitions of the underlying graph and connections between them formed by edges connecting vertices in different partitions) has disconnected components or is very sparse and execution is suffering from alternating skew (skew that is not consistently observed in the same nodes). In the first case, execution of separate components is able to proceed in a completely independent manner and partial final results can be obtained considerably faster than with global synchronization barriers (up to 8x faster in the experiments described in this thesis). In the second case, event execution times can be greatly compressed with observed speedups of up to 1.375. With very densely connected or even completely connected partition meta-graphs, however, partition-level local synchronization quickly loses effectiveness (but does not add any significant overhead either).

Vertex-level local synchronization

While the previous mechanism tracks dependencies uniquely at the partition-level, vertex-level local synchronization extends its behavior by allowing the execution of multiple disjoint computation passes over the vertices of a partition. Thus, even if the dependencies of a partition have not all finished the current superstep, that partition might schedule a partial execution of the next superstep for those vertices that already have all the data they need.

The implementation of the mechanism done in the context of this thesis considers a maximum of 2 execution passes over a partition in a single superstep. When, at the time of scheduling the next execution of a partition, all its dependencies are already finished, a single complete pass is performed as before. If, instead, the percentage of finished partitions lies between a configurable interval, a first incomplete pass is scheduled that will consider only those vertices whose dependency set is contained in the set of partition dependencies already finished. Once all the dependencies actually finish, a second pass, encompassing all the remaining vertices, is then scheduled.

Experiments performed with this mechanism have shown that it is able to obtain performance improvements even in scenarios with high partition meta-graph connectivity where partition-level local synchronization failed. In particular, improvements of up to 16% over the latter mechanism have been observed. With non-favorable executions, however, the 2-pass system needs further optimizations to reduce the overhead.

1.4 THESIS OUTLINE

The rest of the thesis is organized as follows: Chapter 2 explores some of the systems and research related to the topic of graph processing and incremental computation as well as to the specific contributions of the thesis. Chapter 3 provides an overview of RTGiraph, describing the main concepts necessary for its understanding, its main components and relations and an example of incremental graph computation. Chapter 4 and Chapter 5 describe the work done to implement the contributions of this thesis (event pipelining and local synchronization, respectively) along with relevant design decisions and exploration of alternative designs. Chapter 6 tests and evaluates the aforementioned contributions using both artificial and real environments. Chapter 7 concludes with a summary of the thesis and of its main findings.

2 BACKGROUND & RELATED WORK

In this chapter, a brief overview is made over the principle systems and concepts which are necessary for the understanding of the contributions of this thesis or which are related to them and show alternative existing work.

2.1 PREGEL

Pregel [17] is a computational model specifically designed for the processing of very large scale graphs on clusters with thousands of commodity nodes. Developed by researchers at Google, Pregel provides transparent and efficient scalability and fault-tolerance of synchronous graph computations while providing a simple and easy to program API.

Pregel's execution flow is based on the Bulk-Synchronous Parallel (BSP) computational model first introduced by Valiant in 1990 [31]. In this model, a computation engine is composed of components (processes) that process data local to that component; an interconnection network that enables communication between such components through the sending and receiving of messages; and a mechanism that allows the synchronization of all these computational components. Using the three aforementioned mechanisms, computation in a BSP model evolves through a series of supersteps which can themselves be divided in 3 smaller stages:

1. Computation — Processes perform computations on their local data with no interference from other processes.
2. Communication — Data resulting from the computation stage is sent to other processes following the dependencies determined by the application being executed. Messages received by a process are only made visible to it on the following superstep.
3. Synchronization — After having done all computation and communication for the current superstep, a process reaches a global barrier where it will wait until all other processes have finished computation and communication for that superstep.

This situation is represented in Figure 4.

In Pregel, each vertex of the graph corresponds to one of the processes in BSP and the computation is also split in a series of supersteps. In each such superstep, all vertices execute a user-defined function. This per-superstep and per-vertex execution uses as its input

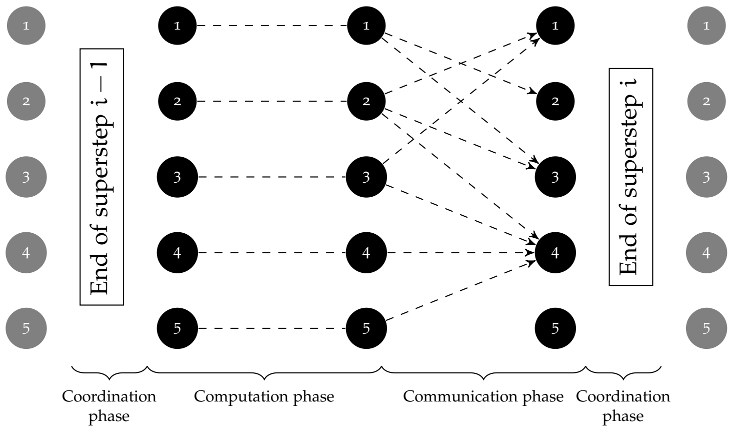


Figure 4: Example of a superstep execution in the Pregel/BSP model with a 5-node cluster.

the value of the vertex in that superstep and the messages sent to it by other vertices in the previous superstep. During execution, the state of each vertex or its outgoing edges can be updated and messages can be sent to other vertices which will then be received in the consequent superstep. When a vertex has nothing else to execute, it can change its state to halted and, unless it receives messages during the following communication stages, it will remain in that state and skip computation until the global execution ends. The end of a global execution occurs when all vertices of the graph are in a halted state.

Because synchronization is only necessary at the end of a superstep, the model is able to scale well in distributed implementations. In addition, its synchronicity makes it easy to develop new graph applications or reuse existing algorithms and eliminates the possibility of deadlocks or data races. Pregel is also useful in that it automatically handles such things as graph partitioning and application distribution over the nodes of a cluster, inter-node communication and coordination and fault tolerance, leaving the developer free to focus his attention on the actual algorithm to be run and not on the management of the cluster.

Although the original implementation of Pregel made by Google was not made public, the Pregel model has been implemented by several open-source projects such as Giraph [27], Hama [28] and GPS [25].

2.2 GIRAPH

Giraph is arguably the most popular open source implementation of the Pregel architecture for graph processing. Written in Java and running as an Hadoop MapReduce job, Giraph integrates nicely with existing Hadoop infrastructure and components such as HDFS. Initially developed by Yahoo, it gained Top Level Apache Project status in 16/05/2012.

Giraph is now being used by several organizations in production environments: Facebook has been using Giraph to power their OpenGraph feature since 2012 [9]; LinkedIn has implemented some of their graph-intensive workflows in Giraph to obtain improved processing speeds [13]; Twitter is also reportedly using Giraph with their dataset. Due to the nature of open source, these organizations have contributed their changes to Giraph, enabling it to grow both in terms of features as in terms of maturity.

2.3 incremental computation

In recent years, the popularity of distributed batch processing tools like Hadoop or Giraph, with their easy to scale implementation, has skyrocketed. Concurrently, cloud-enabled architectures, which facilitate the creation and management of dynamic clusters containing thousands of nodes, are becoming increasingly widespread. These two factors have gradually shifted the focus from improving the latency of a computation by optimizing the amount of computation needed, to increasing the size of the clusters in order to attenuate the effect of the extra computation.

However as organizations increasingly rely on online data mining to provide realtime or close to realtime insight on collected data, the efficiency of these batch processing systems started being questioned.

One of the most frequent criticisms of these systems focused on the fact that each time an analysis was started on a dataset, it did so from scratch. Due to advances in storage technology and data processing mechanisms, it is no longer the case that data is collected, analyzed and then discarded. In fact, most machine learning algorithms usually experience an increase in accuracy as more data is made available to them. Consequently, data mining is now being increasingly done over evolving incremental data sets where the input difference between two different executions corresponds to a set of mutations which might affect only a small fraction of the entire dataset.

Given this realization, researchers started investigating the benefits of adding concepts from incremental computation to these systems. Incremental computation, in a nutshell, focuses on saving the results of intermediate computation steps so that, when an execution with a modified input is requested, only those sections of the data that depend on the modified input are recomputed. [23] provides a vast categorized bibliography related to incremental computation, from implementation techniques to analysis.

One of the first integrations of incremental computation concepts with distributed data processing systems was developed by Google with its Percolator project [22]. With Percolator, Google converted its indexing system to an incremental system, allowing them to process documents as they were crawled and reducing their processing latency by a factor of 100.

Incoop [5], in the area of generic Hadoop MapReduce computations, and Kineograph [8], in the area of graph processing, are other popular examples of the application of incremental computation concepts in large scale distributed data processing systems. Incoop reports 15-30% time savings through the use of their incremental scheduler while Kineograph reports improvements in the order of 50-60% for a small set of graph applications.

2.4 RTGIRAPH

RTGiraph is a system which attempts to apply incremental computation to Giraph, inspired by initial work done with GraphInc [7]. Unlike Kineograph, which designed new application semantics specifically tailored to incremental computation, RTGiraph aims to keep the same application semantics as those used by Giraph, so that Giraph applications can take advantage of incremental computation without requiring any changes to their source code.

The development of RTGiraph started as a fork of the original Giraph project by researchers at Telefónica as part of the grafos.ml project [26]. The computation API and basic configuration code have been kept unaltered so that applications developed and compiled for Giraph can run in RTGiraph without need of recompilation and can, thus, automatically take advantage of the benefits of incremental computation. Chapter 3 details the main concepts and components associated with RTGiraph and incremental graph processing.

2.5 PIPELINING

Embarrassingly parallel computations (with no or very few dependencies) can be easily parallelized by splitting the input in several parts and running them independently of one another. However, as the number of data dependencies increases, finding ways to parallelize execution becomes increasingly harder and inefficient. A common mechanism to obtain some degree of concurrency in these cases is to implement a pipeline system, functioning in a way similar to assembly lines.

A pipeline is regarded as a set of independent processors connected in series in such a way that the input of a processor corresponds to the output of its predecessor. A pipeline execution, consisting of an input given to the starting processor, progresses through the pipeline through a series of pipeline steps until the desired output is obtained from the final processor. By taking advantage of the independence of these processors inside the same pipeline step, one can keep them all concurrently busy handling as many parallel executions as the number of different processors in the pipeline. These executions would be skewed in relation to one another so that, at any one time, each processor is handling at most one such execution. As a result, after a small pipeline-warmup period, new outputs can be generated at every pipeline step whereas before each output could only be generated every x pipeline steps with x being the length of the pipeline.

In computer science, the concept of pipelining started being applied to hardware as part of an effort to improve application performance through the use of instruction-level parallelism [20] [21] [11]. This instruction pipelining was achieved by breaking each instruction cycle into a series of stages such as instruction fetching, instruction decoding, execution or memory access. As software grew increasingly complex, pipelining started being applied at increasingly higher levels of abstraction. Software pipelining, for instance, concerns itself with the identification of opportunities for pipelining the execution of loops [1] [15].

Pipelining is also relevant in systems and architectures that have to deal with streaming input where data is continuously arriving. In these systems, threads (node-level pipelining) or nodes (cluster-level pipelining) are assigned unique functions and data follows a predefined flow through each processor. Apache Storm [19] [30] and Apache S4 [29] are two popular distributed stream processing system and accurate examples of cluster-level pipelining.

The use of pipelining in RTGiraph corresponds to a mixture of node-level and cluster-level pipelining. The RTGiraph pipeline is composed of a series of abstract processors, spread over different threads and nodes, responsible for computing consecutive supersteps of an application execution. The output of the execution of a superstep is fed as input to the execution of the next and several events can be run simultaneously in this conceptual pipeline at different stages of execution (different supersteps).

2.6 LOCAL SYNCHRONIZATION

Coordination among different processing components is an essential component for the execution of any parallel or distributed system. This coordination, also referred to, at times, as synchronization, can be done in different ways. Bertsekas & Tsitsiklis [4, s. 1.4] provide an excellent overview of issues related to synchronization in distributed and parallel systems.

On the one hand, a computation might be divided in a series of phases in which processors operate independently until reaching the end of the phase. Upon reaching this point, they synchronize with their peers before being allowed to continue. These computations are also known as synchronous computations. Because they follow well-defined phases, they are easy to understand, implement and analyze.

In addition, because coordination is done at scheduled intervals, at the end of each phase, the actual coordination logic between different processors is usually simple and straightforward.

On the other hand, computations might be distributed over different processors without the notion of phases, with each processor executing at its own rate and requiring less strict coordination with its peers. Such computations are known as asynchronous. Because they do not require constant synchronization with other processors, these computations can make better use of the available computational resources and spend less time idling while waiting for other processors to respond. However, the downside to this is that the state of a computation at any one point is something which is hard to determine and may, in fact, differ from execution to execution, making the understanding and debugging of these computations a very complex undertaking. In addition, such algorithms need to be designed from the ground up with asynchronicity in mind, being able to operate with partial data from peers and working towards some kind of convergence. Sometimes such a convergence might not be possible or might require complex agreement protocols which results in the use of more time and resources, possibly negating the improvements previously identified.

Synchronous computations can be further divided into 2 different categories depending on the set of peers a processor waits for before continuing to the next phase. The first – globally synchronous computations – require each processor to wait until all other processors finish the current phase before being allowed to continue to the next one. This can be achieved using simple mechanisms such as timeouts on upper computational and communication bounds or through the use of semaphores or barriers accessible by all processors. The second – locally synchronous computations – retains the concept of phases but relaxes the coordination requirements so that each processor only has to wait until it is sure to have received all the information necessary to start the next phase. It does not have to wait for other processors from which it is sure not to receive any data. To achieve this, each processor must know about the messages it has to receive at the end of each phase which, in turn, requires some form of dependency tracking mechanism. While this dependency tracking adds some complexity and possibly overhead to the computation, it is able to make a better use of resources by reducing idle times introduced by global synchronization: if a processor starts lagging behind during a phase, with global synchronization all other processors must wait for this one to finish before being able to continue; with local synchronization only those processors that require data in which the lagging process is working on will wait for it while all the others are able to proceed.

With respect to graph processing, the field is mostly divided between the two opposite sides of the synchronization spectrum. On the one hand, Pregel-inspired systems (Giraph, Hama, GPS, GiraphRT) rely on global synchronization to progress through the execution. On the other hand, systems like GraphLab or PowerGraph focus on asynchronous computations. However, none of these systems explore the concept of local synchronization to attempt to reduce the negative impacts of global synchronization while keeping its benefits, which is precisely one of the contributions of this thesis.

Although no graph processing system actively employs local synchronization in their behavior, some research has been done regarding its application to the generic BSP model on which Pregel is based.

Fahmy & Heddaya [10] were among the first to explore the concept of local synchronization in BSP, replacing the global barriers by what they refer to as lazy barriers. Lazy barriers make use of message counters to decide when processors waiting on that barrier can proceed. The determination of this number of messages can be done in one of 2 ways. If the communication pattern of the application is completely known a priori for each superstep, information can be passed to the executing system informing it of the number of messages a processor will receive in that superstep. If, however, this pattern is not known, it has to be determined at runtime by compiling message sending statistics from all involved processors. The authors in [24] employ a similar message counting mechanism to achieve zero-cost synchronization, obtaining latency improvements of 5% to 18%.

Kim et al. [14] also study the application of local synchronization to the BSP model but do so by completely removing the barriers at the end of each superstep. In its stead, the semantics of the read operation are changed to block such operations on regions that still have not received as many writes as expected. Therefore, processors immediately proceed to the following superstep and only block when they attempt to access an area which still has not received all the information from the previous superstep.

Although the aforementioned research on local synchronization was done in the context of the BSP model, it did not consider the specific case of graph processing, where dependencies can be considerably more common and irregular than with other types of structured data. In addition, on those where a prototype system was actually implemented, they were not tested in multi-node (clusters or cloud) environments as used today but in single-node multi-core machines. This necessarily results in a difference in communication time which might or might not have an impact on the registered improvements with local synchronization. Furthermore, they all rely on changes to the programming model, something that RTGiraph attempts to circumvent.

A more recent paper [34] describes a generic framework which strives to solve the problem of global synchronization in the presence of high network jitter registered in cloud environments. This is achieved through a combination of 3 mechanisms. The first, local synchronization barriers, examines information provided by the application to identify the read and write dependencies of queries to determine when processes can continue to the next tick. The second, dependency scheduling, recognizes that even with local synchronization barriers, jitter might still lead to long idle times. To solve this, partial executions of the next tick are scheduled using the currently available data. Only the subset of tuples that already have all the necessary data to compute the next tick will be processed. The remaining ones will be processed in secondary execution schedules as more messages from the previous tick are received. The third makes use of data and computation replication to attempt to counteract the effects of jitter by providing multiple sources for the same data and thus decrease the probability of observing long message communication times. The partition-level and vertex-level local synchronization modules designed and implemented in this thesis are very similar to the two first mechanisms described in this paragraph, respectively, local synchronization barriers and dependency scheduling. However, as RTGiraph aims to maintain the existing programming model, dependency information is extracted from the structure of the graph instead of being directly provided by the application.

3 REALTIME GIRAPH

Realtime Giraph (RTGiraph) is a fork of Giraph started by researchers at Telefónica with the objective of creating a graph processing system supporting incremental computation features while, at the same time, maintaining an API compatible with that of the original Giraph system. By ensuring this compatibility, Realtime Giraph strives to provide hassle-free transitioning of algorithms from one system to the other without having to change the source code of the applications.

While the API is largely shared with the original project, much of the backend code was modified to enable a clearer separation of responsibilities among the different components. This chapter will provide a summarized view of each major component of the system and show how those components orchestrate the execution of applications over graphs, while taking advantage of the results of previous computations. The understanding of these concepts will enable the reader to better comprehend the current limitations of the system and provide useful context to better appreciate the implemented optimizations.

3.1 EXAMPLE

In this section, an example execution of RTGiraph is shown with the objective of motivating and introducing context for the description of the concepts and modules that will follow in subsequent sections. This example will look at the execution of Single Source Shortest Paths (as described in Section 6.4.2.1) over 2 events (graph mutations) of the graph shown in Figure 5. The considered source vertex is vertex 1.

Figure 6 shows an overview of an execution corresponding to event 0. By definition, event 0 corresponds to the initial execution and, as such, is no different from an execution from scratch over the base graph. To obtain the final results, a total of 14 vertex computations have to be made: 2 for vertex 1, 3 for vertices 2 and 4, 4 for vertex 3 and 2 for vertex 5.

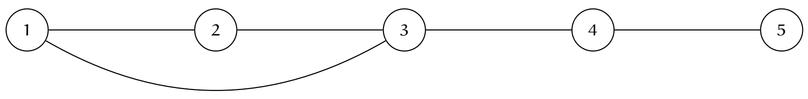


Figure 5: Base graph used in the example

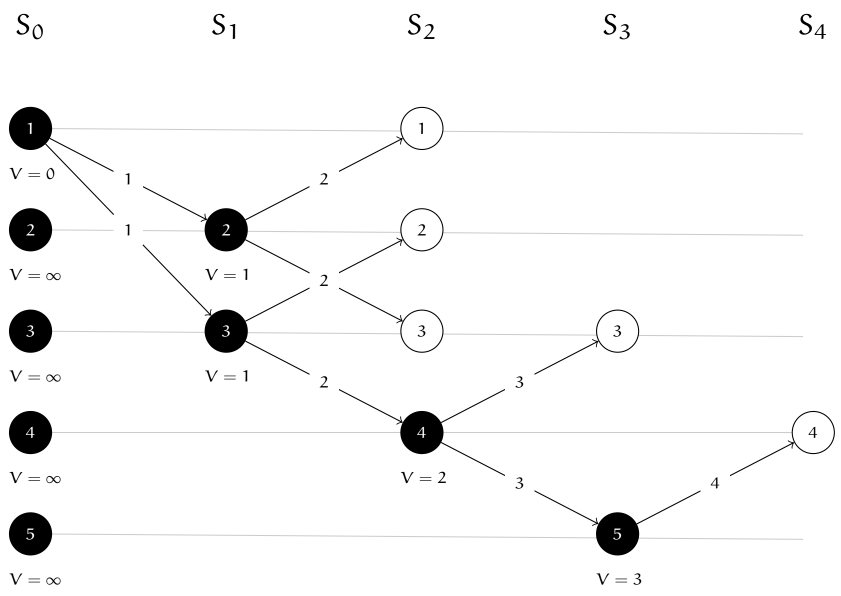


Figure 6: Example execution for event 0 (from scratch). Black vertices represent computations leading to updated values; white vertices represent computations leading to unchanged values. Full edges constitute actual messages sent due to a computation.

After event 0, a new edge is added between vertex 1 and vertex 5, forming event 1. A recomputation from scratch, as would be done by Giraph, would now take 13 computations to arrive at the conclusion that the shortest path from vertex 1 to vertex 5 now has a length of 1 instead of 3.

With RTGiraph, however, that is not the case. Figure 7 showcases an execution of event 1 with RTGiraph. In superstep 0, those vertices affected by the mutation of event 1 are executed. In this case, those would be vertices 1 and 5. After computation, both vertex 1 and vertex 5 finish with exactly the same state as in the previous event with the exception that an extra message containing the value 1 was sent through the $(1,5)$ edge. This message is added to the new state for that superstep and transferred to the next one as delta information. In this first superstep, a total of 3 computations were saved by reusing the stored state for vertices 2, 3 and 4 which saw no change.

In superstep 1, vertices 2 and 3 reuse the state of superstep 1 for the execution of event 0 as there has been no change to their initial values from the previous event. Vertex 5, however, now has received the message from vertex 1 so it computes, sets a new vertex value of 1 and sends a message with value 2 to vertices 1 and 4. The new message and updated vertex value are then added to the state of superstep 1 and passed on to the next superstep as delta information. In this superstep, by skipping the recomputation of vertices 2 and 3, a total of 2 computations were saved.

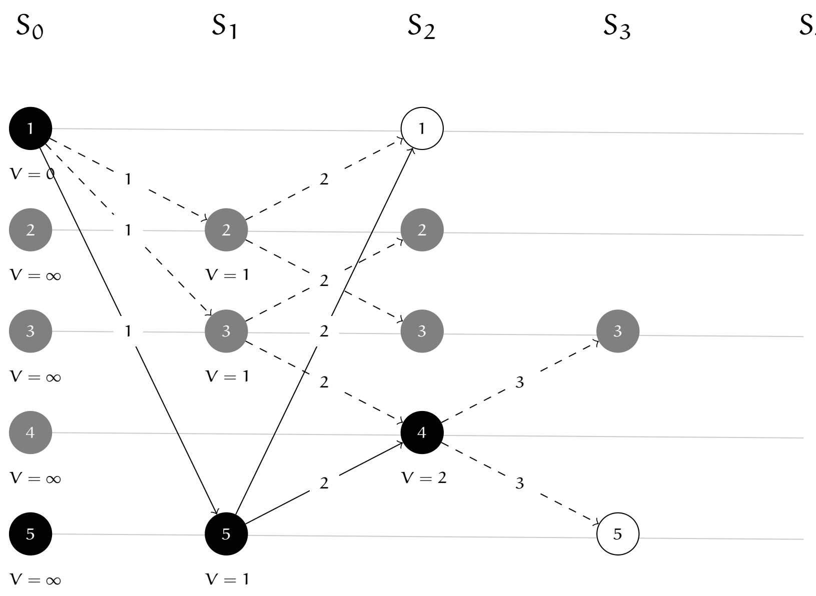


Figure 7: Example execution for event 1 (using memoization). Black vertices represent computations leading to updated values; white vertices represent computations leading to unchanged values; gray vertices represent computations saved due to memoization. Full edges constitute actual messages sent due to a computation; dashed edges constitute messages saved due to memoization.

With superstep 2, vertices 2 and 3 once again reuse previous state and vertex 5 does not compute as it is halted. Vertex 1 receives a new message from vertex 5 so it computes but keeps the same final state. Vertex 4 also computes due to the new message from vertex 5 (which is aggregated to the stored message from vertex 3) but since its final value in this superstep is still 2, as in the previous event, a new message is not sent along its outgoing edges. The delta for vertex 5 coming from superstep 1 continues to be propagated to the next superstep. A total of 2 computations are saved.

Superstep 3 starts with the execution of just vertex 5 due to the stored message from vertex 4 in the state from the previous event and the changed value from the delta. Because the current value of vertex 5 is already lower than the value of the message, no message is sent and vertex 5 halts. In the previous event, vertex 5 had sent a message with value 4 because its value had been updated to 3 in superstep 3. As no message is sent this time, a negative message is added to the state of this superstep to cancel the stored one and is also appended to the delta information passed on to superstep 4. This delta information also continues to propagate the updated value of vertex 5. No change is made to the input of vertex 3 and, therefore, memoized state for this vertex is used, saving 1 computation.

In superstep 4, because the only stored message in the state has now been canceled by the negative message in the delta, vertex 4 continues to be halted and, therefore, every vertex is halted, marking the end of the execution of the event. The final result for the event is composed from the updated values contained in the delta information: value 1 for vertex 5. This result constitutes a delta from the results of the previous event: all vertices that do not show in the result for an event keep the same final value they had in the previous event.

By using memoization and incremental computation concepts, a total of 8 computations (and associated sending of messages) were saved, a figure which represents more than half of the total number of computations needed for an execution from scratch.

3.2 CONCEPTS

This section aims to describe the main concepts related to the organization and execution of applications over graphs in RTGiraph as exemplified in the previous section.

3.2.1 *Graph*

RTGiraph, just like the original Giraph can handle both directed and undirected graphs described in a great variety of formats.

After being loaded according to the defined input format, all graph-related data is stored in memory-friendly byte arrays which are iterated over during the computation using object reuse (the same object instances are used throughout the iteration, just their data changes). While this makes data management and storage more complex, it saves significant memory and reduces the amount of time spent by the JVM creating and destroying new objects which results in a significant performance improvement over traditional storage of data directly with Java objects.

3.2.2 *Partition*

Graphs used in real-world environment are becoming significantly larger. It is not uncommon to find graphs weighting several hundred gigabytes or even terabytes. As such, storing the entire graph information in each node is not only a waste of resources (as usually each node will only process a subset of the graph) but also an increasingly difficult proposition.

To tackle this problem, the graph is divided in a set of partitions. Both in Giraph and RTGiraph, each worker node is capable of handling multiple partitions simultaneously. Since execution (and most of the interaction with the graph) is handled and scheduled at the partition level, supporting multiple partitions per node situates the granularity level of the management at a configurable level between that of vertex-level management (equivalent to one partition per vertex) and graph-level management (equivalent to a single partition per graph).

3.2.3 *Applications*

One of the objectives of RTGiraph is to support the concurrent execution of different applications over the same base graph and events in an incremental manner. Therefore, most of the data managed by Realtime Giraph uses application identifiers as part of their index key to keep the executions of different applications over the same data from tampering with each other. In this way, applications can run in an independent manner, not limited by the rate at which other applications are currently processing the graph.

3.2.4 *Events*

Events are the concept that most distinguishes RTGiraph from the original project. In Giraph each execution is handled independently from previous executions. When the input graph changes, a recomputation of the graph is done from scratch. While this might be acceptable if inputs represent totally different graphs, in the case where inputs represent mutations of the same graph, it constitutes a serious waste of resources. RTGiraph, by saving intermediate execution data (memoized state), allows the reuse of previously calculated results. This is accomplished by not running applications directly over the graph or its partitions but over events in the graph.

An event represents a mutation in the base graph to which it is applied. One might consider event 0 to be the event of adding all the vertices of the graph along with their connecting edges, that is, the event that from nothingness creates the base graph. Since this is the initial event and no data was stored, executing it corresponds to a normal execution just like the one done by Giraph. If an edge were to be removed, the resulting graph could be easily constructed from the graph in event 0 by applying the specified edge removal. The removal of this edge would then represent event 1 and the resulting graph would be the view of the graph from the viewpoint of event 1. Execution of event 1 could now take advantage of data recorded during execution of event 0 to speed up and skip parts of the execution of event 1. An example of this is given in Section 3.1.

An important notion to bear in mind is that, because RTGiraph supports several simultaneous applications, events cannot directly modify the base graph as an application may want to be able to obtain a view of the graph at a particular event which is not necessarily the one being considered by other applications. Therefore, event information is actually stored as a delta from the previous event and the final view of the graph for that event is the result of applying, in sequence, the deltas from all events since event 0. From time to time, the graph may be rebased (set a new event as event 0) when previous event data is sure to not be necessary. By doing this, the chain of deltas to apply is kept at a manageable level.

3.2.5 *Supersteps*

Supersteps represent computation rounds done while executing an event. In each superstep, the partitions containing vertices to be computed are submitted for execution; new state and deltas are generated; and messages are sent and received from other partitions.

The advancement from one superstep to the next is synchronized across the whole set of cluster nodes participating in the active event. Only when all the nodes have finished processing the current superstep are they allowed to progress to the computation of the following. This is ensured by the use of two global barriers: one at the start of the superstep and one at the end.

The final superstep of an event is the one where all the nodes are inactive. In other words, a superstep where all partitions have no work to be done and, thus, all vertices have halted, represents the final superstep and the final result of the execution can be obtained by looking at the generated delta of the previous event.

3.2.6 *Node status*

Nodes that compose the RTGiraph cluster might be in one of 3 states:

1. Not involved — Nodes may not be involved in an event if its current execution has not affected any of the vertices contained in the partitions local to the node. Non-involved nodes have no knowledge about the event and, thus, have no event executors associated with it nor do they participate in the advancement of supersteps.
2. Involved — Nodes where at least one vertex was involved in the computation of an event are considered as being involved in that event and, therefore, have an associated event executor and participate in the advancement of supersteps.
    a) Active — A node is active if it has work to do in the current superstep, that is, if any of its local partitions have non-halted vertices that may need recomputation.
    b) Inactive — A node is inactive in a superstep if all its vertices are in an halted state in that superstep.

Transitions from non-involved to involved states happen when an involved node sends a new message to one of the vertices in the non-involved node. When this message is received by the non-involved node, it notices that it comes from a previously unknown event; creates the necessary components to handle it (in essence, the event executor) and processes it. When a node becomes involved in an event, it remains involved until the end. Transitions between active and inactive states can happen in both directions and are motivated either by the set of messages received by a partition and/or by set of vertices that is halted in that particular superstep.

3.2.7 *Memoization*

Memoization is the concept of saving intermediate results and state of computation to allow the reuse of information in recomputations of part of the main graph. In the memoization process, 2 main memoization data structures are considered: state and delta.

3.2.7.1 *State*

Memoized state contains information about the state of all vertices of a partition in a certain superstep. At the end of each superstep, and for each vertex, the following information is stored:

- *haltedIn* — A boolean representing the halted state of the vertex at the beginning of the superstep.
- *valueIn* — The value of the vertex at the beginning of the superstep.
- *haltedOut* — A boolean representing the halted state of the vertex at the end of the superstep.
- *valueOut* — The value of the vertex at the end of the superstep.
- *messageOut* — The message sent by the vertex during the current superstep (if any).

In addition, for each partition, the collection of messages received during a superstep from both local and remote partitions are kept in a mailbox.

This memoized state is used by the event following the current one to understand the base state of the vertices in the associated superstep and what results were propagated to the next one. This allows the following event to skip computation of vertices where their initial superstep conditions have not changed.

3.2.7.2 *Delta*

Memoized deltas contain information about the changes in state in the current event when compared to the same superstep in the previous event. At the end of each superstep, the following information is stored:

- *deltaComputeState* — For each vertex whose initial state now differs from the one stored in the memoized state for the previous event, a structure is stored containing the new vertex value and/or the new vertex halted state.
- *deltaMessages* — During execution of a new event, a message that had been sent in a certain superstep of the previous event might not have been sent in the current one. In addition, during execution of the current event, more messages could have been sent that had not been sent in the previous event. Information about this change in the messages sent between supersteps is stored in the *deltaMessages* collection.
- *toRun* — Instead of having to reanalyze the whole memoized state and delta to determine which vertices should attempt computation at the current superstep, that collection of vertices is populated upon the execution of the previous superstep where that determination can be done in a less complex way. The set of vertices to attempt computation at the next superstep is then stored in the *toRun* structure.

Memoized deltas are used by the following superstep in the current event to determine what changed from the state read from the previous event for that superstep. Only vertices that have experienced a change in their inputs are recomputed and, based on the results of that computation, the delta for the next superstep is created.

3.2.7.3 *Storage and access pattern*

Memoized state and delta are stored in separate files, one per superstep. At the beginning of a superstep, RTGiraph reads the whole file for that superstep, obtaining the base state and, if it exists, a delta generated by the previous superstep. After applying the delta to the base state, the new state for the current superstep is created and those vertices which have received messages and/or are contained in the *toRun* set are executed.

After execution, the updated state for that superstep, after application of the deltas and the new values generated by the computation, is written to the file of that superstep. The delta is written to the file of the next superstep, where it is to be applied during its execution.

This 1 read/2 write access pattern is exemplified in Figure 8.

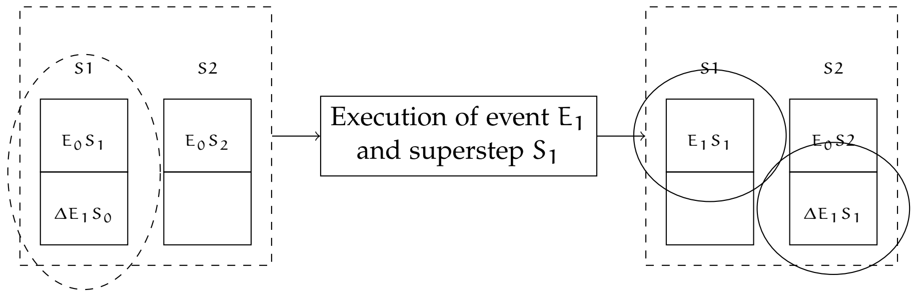


Figure 8: Example of the normal memoization read/write pattern. Event 1 executes superstep 1. The inside of the dashed boxes represent partial snapshots of the filesystem before and after the execution of the superstep. Solid ellipses represent data writes and dashed ellipses represent data reads.

3.3 MAJOR COMPONENTS

Having looked at the main concepts of importance to RTGiraph, this section describes the most important components and the way in which they are connected to one another.

3.3.1 *Local master*

The local master is the master component at each node and acts as a frontend for that node, both to the client and to other nodes in the system. Its major responsibilities are:

- Bootstrap the remaining major components on the node.
- Establish connections to other nodes in the cluster.
- Listen for new event requests from clients and route them to affected nodes. If that node is not involved in the event, an associated event executor is created and managed.
- Listen for new remote deltas from event executors in other nodes. These remote deltas are routed to the corresponding local event executor or, in the event that such an executor does not exist, a new executor is created to handle it.
- Listen for event finish notifications and write to disk the results generated by local event executors.

3.3.2 *Global sync*

Global sync is the component that coordinates the execution of different events and, for each event, the execution of its supersteps. There is a single global sync entity per cluster and communication between it and the other nodes is done indirectly through ZooKeeper.

Each event executor, when being activated, reaching a start barrier or an end barrier, writes the type of event along with its current status (active or inactive) to ZooKeeper. The global sync module configures ZooKeeper listeners to listen to these changes and updates its current view of the cluster which is mainly composed by 3 mappings:

- Mapping of event contexts to active involved nodes in an event, along with the current superstep at which their event executor currently is.
- Mapping of event contexts to active involved nodes that are currently doing computation, that is, haven't yet reached the end barrier for their superstep execution.
- Mapping of event contexts to inactive involved nodes in an event, along with the current superstep at which their event executor currently is.

With these three mappings and information about the type of event just received, the global sync module is able to determine when all nodes are waiting at the same barrier and, thus, when it should be broken or when an event has finished:

- If a *start barrier reached* event is received; the start barrier has not been broken yet; every other involved event executor in the cluster is in the same superstep; and there is at least one active node then the start barrier can be broken.
- If a *start barrier reached* event is received; the start barrier has not been broken yet; every other involved event executor in the cluster is in the same superstep; and all the involved nodes are inactive then the event has finished.
- If an *end barrier reached* event is received; the start barrier has not been broken yet; every other involved event executor in the cluster is in the same superstep; and there is at least one active node, then the start barrier can be broken. This may happen because inactive nodes in a superstep jump directly to the end barrier without passing through the start barrier as shown in Section 3.3.6.
- If an *end barrier reached* event is received; the start barrier has been broken; every other involved event executor in the cluster is in the same superstep; and there is no node currently doing computation, then we end barrier can be broken.

Notifications of start/end barrier breakages or event completion are also communicated back to the individual nodes in the cluster indirectly through ZooKeeper using special ZooKeeper nodes dedicated to that particular event.

3.3.3 *Local sync*

Local sync is the component residing in each node responsible for interacting with ZooKeeper and, thus, indirectly with the single global sync. Local sync routes status updates from event executors to ZooKeeper and listens for changes made by global sync to ZooKeeper nodes associated with all events that node is involved in. These changes are then passed on to the respective event executors allowing their advancement/termination.

3.3.4 *Async state backend*

The async state backend module is a per-node component which allows the management of most data associated with the graph, the cluster and the applications. In particular, it:

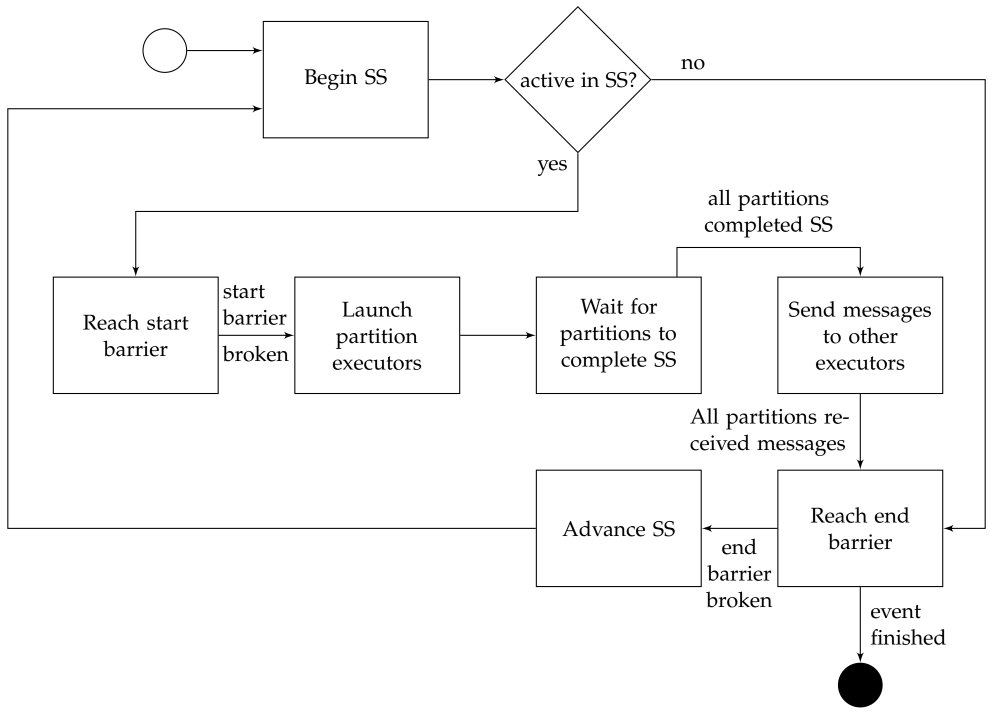
- allows the reading of graph partition structures containing the set of vertices and edges that form that partition.
- manages a partition directory service which records associations between all partitions of the graph and the nodes in the cluster where they are located.
- handles requests for accessing and updating memoized information from executions of supersteps of the same or previous events.

Therefore, the async state backend serves as a central data directory for all the data needed to ensure correct execution of graph applications on each node.

3.3.5 *Partition executor mediator*

The core of the work done by RTGiraph involves the execution of partitions for each superstep, that is, the execution of the provided application on every vertex of the partition for each superstep of the running event. If this were done independently by each event executor, managing and monitoring allocated resources to this expensive task would require complex coordination.

The partition executor mediator is a component that aims to solve the problem described in the previous paragraph. Through the use of a thread pool of configurable size and a queue of partition executors, this mediator is able to ensure that, at any one time, the number of concurrent partition executors is within the specified limits and, thus, the impact of their execution on the system resources is controlled.



Begin SS
active in SS?
no
yes
all partitions completed SS
Reach start barrier
start barrier broken
Launch partition executors
Wait for partitions to complete SS
Send messages to other executors
All partitions received messages
Advance SS
end barrier broken
Reach end barrier
event finished

Figure 9: Flowchart of the basic event executor execution flow. Blank circle symbolizes start of flow, filled circle symbolizes end of flow.

3.3.6 *Event executor*

With RTGiraph, executions of an application over a graph or over mutations of that graph can be done on-the-fly and incrementally through the concept of events. Since each event essentially corresponds to a new execution of an application, they are associated with their own sets of supersteps and intermediate state. Handling the execution flow and data dependencies between supersteps in a centralized module would be unnecessarily complex and confusing.

By using event executors, all information and management needed to ensure progress between supersteps in the execution of an event is kept in its own independent module. An event executor will, based on messages and notifications routed to it by local sync and local master, determine how to advance the execution, reach barriers, create partition executors and send messages. Figure 9 gives an overview of the basic algorithm followed by each event executor.

Handling of notifications from local master or local sync and messages from other event executors are abstracted away in the flowchart as an event executor continuously checks its mailbox for new messages throughout the entire flow. Also abstracted in the flowchart is the management of resources: only a limited number of partition executors are launched simultaneously to control the amount of memory and computation used by the execution. When a partition finishes computation and there are still partitions in the launch queue, one of them is removed from the queue and launched to take the place of the just finished partition.

3.3.7 *Partition executor*

Partition executors are the components that perform the actual work. These executors iterate over all the vertices that have to run for a superstep (either because they received a message, they have changed state from the previous event, they have never run before or they haven't halted) and execute the provided Giraph computation on those vertices, accumulating updated vertex values and messages in new state and delta objects which are saved to disk; and accumulating the messages also in an outbox which is passed on to the event executor for sending to other partitions.

Relying on memoized state, partition executors are able to identify if the initial conditions before computation of a vertex differ from those recorded for the same vertex in a previous event. If no changes were detected, computation of that vertex is skipped and the memoized results stored in the state object are used instead. If, however, a change is detected, that vertex is recomputed in the new environment; the state object is updated with the new values for use by the next event; and these new values are added to a delta object passed on to the execution of consequent supersteps of the same event to overwrite the state data read from a previous event.

3.4 ARCHITECTURE

Figure 10 shows how the major components described in the previous section are organized and interact with one another to allow the functioning of the system as a whole.

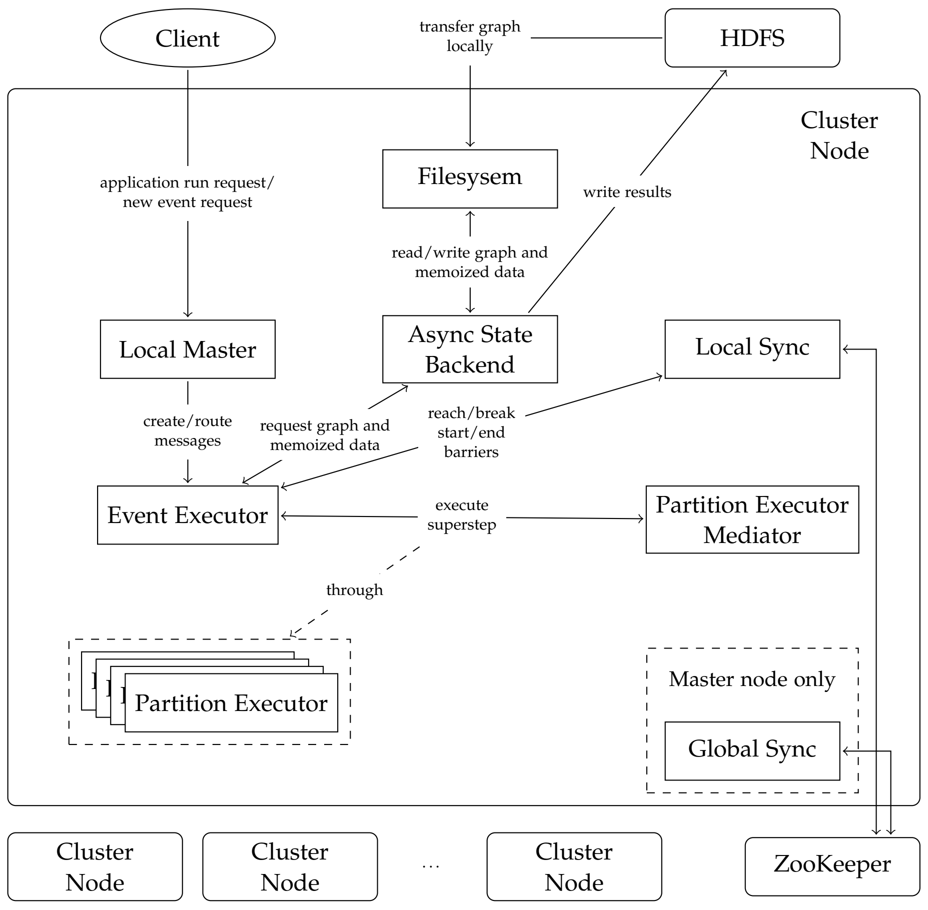


Figure 10: Architectural organization of the major components. Rectangles represent RTGiraph/system components, rounded rectangles represent cluster entities and ellipses represent actors.

4 EVENT PIPELINING

4.1 INTRODUCTION

RTGiraph has the capability of processing a series of events representing mutations of the initial graph. Currently, these events are processed one at a time, in order, reusing the memoized information of previous executions. While this, in itself, already provides significant improvements over the standard Giraph execution (where subsequent computations start from scratch), it still has room for improvement. In particular, by exploiting the pattern of access to the memoized state, several events can be processed concurrently in a pipeline, greatly decreasing the necessary time to process them all. This is shown in Figure 11.

Consider the information required for the execution of a certain partition in superstep S_X of event E_Y, as described in Section 3.2.7: the state of that partition in the same superstep (S_X) on the previous event (E_{Y-1}); and the deltas from the execution of a previous superstep (S_{X-1}) in the same event (E_Y). The former represents the state of the vertices in the partition as recorded by the previous event while the latter describes changes in value that have been registered during the execution of the current event and which, thus, overrule the memoized state. Since deltas are information internal to the execution of an event, they do not influence in any way the execution of other events. State however, since it's read from information written by a previous event, does introduce dependencies.

Figure 12 represents, in a graphical manner, the state dependencies discussed in the previous paragraph over the course of the execution of 2 events in the same partition. Looking at the figure, it becomes clear that event E_Y does not have to wait until event E_{Y-1} completely

E_1S_0	E_1S_1	E_1S_2	E_1S_3	E_1S_4	E_1S_5			
	E_2S_0	E_2S_1	E_2S_2	E_2S_3	E_2S_4	E_2S_5		
		E_3S_0	E_3S_1	E_3S_2	E_3S_3	E_3S_4	E_4S_5	
			E_4S_0	E_4S_1	E_4S_2	E_4S_3	E_5S_4	E_5S_5

Figure 11: Pipelined execution of 4 events: E_1, E_2, E_3 and E_4. Assuming ideal conditions and that the execution of each superstep takes the same time of X seconds, this pipelined execution takes 9X seconds against 20X seconds of a sequential execution.

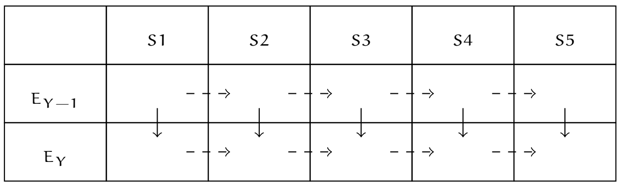


Figure 12: Data dependencies for the execution of 2 events (E_Y and E_{Y-1}) in consecutive supersteps. Solid lines represent state dependencies, dashed lines represent delta dependencies.

finishes its computation over all supersteps but can start computing a superstep as soon as E_{Y-1} has finished computing that same superstep. This is true because, as soon as E_{Y-1} finishes computing superstep S_X, it writes the new definitive state of the graph at that superstep. Since the availability of this new state is the only requirement for E_Y to be able start processing superstep S_X, one can execute the 2 events in parallel at different supersteps as shown in Figure 11.

In order to achieve this, several changes had to be made to the global synchronization module, the local master and to the memoization mechanism. These changes not only add the capability for several events to execute simultaneously, they also guarantee that these simultaneous executions do not affect the correctness of the application. They are described in detail over the next few sections.

4.2 DESIGN AND IMPLEMENTATION

In this section, the major changes required for the implementation of event pipelining are described along with relevant design decisions supporting those changes. In Section 4.2.1, the new event scheduling and synchronization mechanism is detailed, explaining how the global synchronization module decides on whether to allow an event to proceed with execution or not. In Section 4.2.2, the new thread pool and routing mechanism needed to manage several event executors per node are highlighted. Finally, Section 4.2.3 details the changes made to ensure that concurrent events in the same node do not interfere with the memoized state of one another in an illegal manner.

4.2.1 *Global synchronization module*

As seen in Section 3.3.2, the global synchronization module is the entity responsible for advancing the execution of applications and their events by coordinating the breaking of superstep barriers and determining when events have finished executing.

To support pipelining, this module has to relax its requirements for breaking the start barrier for a certain event in a certain superstep.

Previous to pipelining, the requirements for global sync to allow the breaking of a start barrier in superstep S_X by an event E_Y were as follows:

1. E_{Y-1} must have finished computing all supersteps.
2. E_Y must have finished computing superstep S_{X-1} or $X = 0$.

With pipelining, these requirements have to change only slightly to accommodate the changes described in the last section:

1. E_{Y-1} must have finished computing superstep S_X.
2. E_Y must have finished computing superstep S_{X-1} or $X = 0$.

In order to enforce the change in requirement 1, global sync now has to maintain a mapping of all supersteps currently being executed by each event as well as a set of those events which have already finished. When an event E_Y attempts to start a new superstep S_X (in the remainder of this section this shall be referred to as $E_Y S_X$), this mapping is consulted to find the superstep at which E_{Y-1} is. There may be 3 different outcomes of this lookup:

1. E_{Y-1} is present in the mapping and its current superstep is lower or equal to S_X — execution of $E_Y S_X$ is delayed.
2. E_{Y-1} is present in the mapping and its current superstep is higher than S_X — execution of $E_Y S_X$ is allowed.
3. E_{Y-1} is not present in the mapping:
   a) E_{Y-1} is present in the set of finished events — execution of $E_Y S_X$ is allowed.
   b) E_{Y-1} is not present in the set of finished events — execution of $E_Y S_X$ is delayed. This happens because the global sync module does not yet know about the existence of the predecessor event (which might happen if events are not delivered to the global sync by different processes following causal ordering).

When execution of $E_Y S_X$ is delayed, $E_Y S_X$ is added to a waiting map using as index the identifier of the previous event. When the previous event finishes execution of its current superstep, this waiting map is consulted and waiting executions are reactivated.

4.2.2 *Local master*

Local master, as described in Section 3.3.1 is the component responsible for bootstrapping the other core components of RTGiraph in each node that will participate in the execution. It is also the component that will receive notifications of new events directly from the client or from the node chosen as the event group leader as well as remote deltas originated at event executors in other nodes.

Without pipelining, a local master need only manage a single event (and thus a single event executor) at any one time. When a new event is received it will either spawn an event executor (if none existed at that time and the receiving machine would be a participant in that event) or queue the event until the current one finishes. Any remote deltas received would be directly relayed to the active event executor and completion notifications are also clearly associated with the running event.

With pipelining, however, a local master will need to manage multiple simultaneous events and their associated executors. The single thread used for the execution of the event executor without pipelining is no longer sufficient and now requires a thread pool with as many threads as the number of events to be simultaneously processed (that is, the size of the pipeline). Remote deltas now also require routing information identifying the event that generated them so that the local master is then able to direct them to the correct executor. The same happens with event completion notifications.

While the actual implementation of these changes is rather straightforward — include an application context in every delta sent between different machines and record a mapping of application contexts to event executors on the local master — there is an important implementation detail to consider when it comes to the management of the thread pool and sending of notifications of new events to the cluster nodes. Since, by design, each executor is handled in a thread of its own, one has to ensure that the available threads in the thread pool (whose number is fixed by the size of the pipeline) are assigned to the events in the right order. Therefore, if a server is involved in events E_1, E_2, E_3 and E_4 and has a fixed pipeline size of 3, the execution of E_4 has to be queued until one of the threads handling E_1, E_2 or E_3 is freed. Failure to do so (for example, allocating the threads in the pool to E_2, E_3 and E_4 perhaps because this machine only gets involved in E_1 at a later superstep S_8 and is thus only informed of this event at a later point) would result in a deadlock as event E_2 would stop at S_8 waiting for the completion of S_8 by E_1 which cannot possibly complete because it cannot acquire a thread.

To resolve this problem, 2 solutions were considered:

1. Use a fixed-size thread pool as described but inform every node in the cluster of an event as soon as that event is received by the cluster (whether or not a node will actually participate in this event) and as soon as it is completed. By receiving these notifications, each node can schedule the attribution of threads to

events sequentially, following a sliding window with size equal to the pipeline size.

2. Use an unlimited-size thread pool and only inform nodes of an event when they have to participate in their execution. Because a virtually unlimited supply of threads is now available to executors, strict ordering enforcement is no longer required at the local master level. Event executors will be created for each event in which the node is involved and assigned a different thread. The size of the active part of the pipeline is then enforced at the global synchronization module level by adding a third requirement to the two described in the previous section which prevents execution of a superstep by an event if that would increase the number of actively running executors beyond the pipeline size.

The first solution allows for a stricter control of resources by preventing the allocation of a potentially big amount of threads and the creation of event executors and accompanying data structures before they can actually be used. However, in order to accomplish this, it requires extra synchronization logic and the global notification of new events which has additional communication and implementation cost. The second solution is able to keep the number of messages to a minimum by only notifying nodes of an event when those nodes are actually involved in their execution. It also reuses the synchronization logic already available in the global synchronization module. To do so, however, it preloads executors that might only be able to do actual work at some later point in time.

Given that the actual data allocation required per executor when not doing any active execution is minimal in size (a couple of counters, per-partition and per-superstep map initializations); the fact that this allocation would have to be done anyway at some point in time; and because it allows the reuse of existing synchronization code; the second solution was adopted in the implementation.

4.2.3 *Memoization mechanism*

Without pipelining of events, the memoization mechanism worked as described in Section 3.2.7, with state and deltas being stored in one file per superstep and per partition. When reading a partition from disk, the file for that partition and superstep was read to obtain the stored state and delta. The delta would then be applied to the state and, after computation, the updated state would be written back to the same file with an empty delta (since whatever deltas were there initially have now been applied to the state). In addition, the delta generated during the computation of that superstep was appended to the file of that partition but subsequent superstep. Since at any one

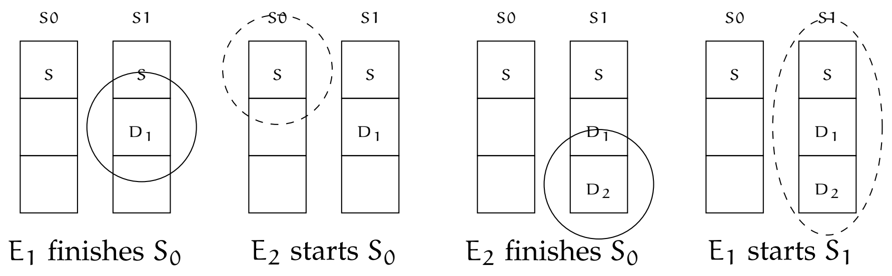


Figure 13: Violation of correctness using the previous memoization mechanism and pipelining. Execution of E_1 is influenced by E_2. S represents state information and D_X represents delta information originating from E_X. Solid ellipses represent data writes and dashed ellipses represent data reads.

time only one event is running, whatever data that event reads is sure to have been produced by the execution of that event or the previous one and so correctness is ensured.

When starting to allow the parallel execution of events, the mechanism described in the previous paragraph does not necessarily ensure correctness of execution. To understand why, imagine 2 events: E_1 and E_2. For simplicity, the concept of a partition will be abstracted, allowing the consideration of whole graph computation in each superstep (and thus considering a single memoized state/delta file per superstep). Figure 13 represents a situation in which E_1 executed superstep S_0 and updated the file for S_0 with the new state required by E_2 and the file for S_1 with the generated delta. E_1 has now moved on to the execution of S_1 and so E_2 is free to execute S_0. Due to delays in the breaking of barriers by the global synchronization module or due to thread scheduling quirks, it is possible that E_2 finishes execution of S_0 and updates the files for S_0 and S_1 before E_1 even reads the file for S_1. If this were to occur (and it did occur in some experiments), then E_1 would read the file for S_1 which now contained a delta introduced by E_2. Therefore, E_1 would act based on data introduced by a future event which clearly represents a violation of correctness properties.

4.2.3.1 *Solution 1*

One way to fix this problem would be to tag the deltas appended to the superstep files with the event that generated them. This would then require that when reading deltas from a superstep file, those that are not compatible with the event currently being executed be filtered out. While this would solve the problem, it is not an ideal solution as it would introduce extra complexity to the reading and writing operations and would result in the reading of a potentially large amount of unneeded data.

4.2.3.2 *Solution 2*

Another possible solution would be enforcing a bigger delay in the pipeline. Instead of allowing E_2 to execute S_0 as soon as E_1 finished executing it, one could require E_2 to have to wait until E_1 finished executing S_1 too. By doing this, the execution of S_0 by E_2 (and consequent modification of the file for S_1) would not be able to influence the execution of S_1 by E_1 as E_1 would already be executing S_2. While this proves to be a simple solution, it does have the disadvantage of reducing the effectiveness of the pipeline. Assume, for simplicity, that each the execution of a superstep in the pipeline takes a fixed time of Y seconds. With X concurrent events all executing for the same number of supersteps, the new pipeline execution schedule would take an extra $(X-1) * Y$ seconds when compared to the original one where events were separated by a single superstep.

4.2.3.3 *Solution 3 - Adopted*

A third solution, and the one adopted in the project, allows for optimal pipeline performance and does not introduce significant extra complexity or wasted time processing unneeded data as happens with the previous two solutions. It consists in having the memoized state and deltas written in different files indexed by partition, superstep and event. In doing so, files will only be modified by the events that own them and the possibility of introducing the kind of conflicts described above is therefore eliminated.

However, using this new memoization mechanism requires some changes to the reading and writing IO pattern. With the original mechanism (and, once again, abstracting the individual partitions), an event E_Y executing superstep S_X did a single read IO from file S_X, obtaining the state from the previous event and the deltas from the current event. After computation, the same event would then do 2 write IOs: one to file S_X, updating the stored state to reflect the new state after computation; and one to file S_{X+1}, writing the generated deltas for the next superstep.

To continue having a single read IO with the new per-event files, this would require that an event E_Y, after execution of S_X, write the new state to the file of the next event in the same superstep ($E_{Y+1}S_X$) and the deltas to the file of the same event but next superstep (E_YS_{X+1}). While this IO pattern sounds feasible, it fails when taking into account the fact that some machines might not be involved in an event from the beginning (or at all). If this were to happen and a machine were only involved in events E_Y and E_{Y+2}, the state needed by E_{Y+2} in superstep S_X would actually be located in E_YS_X, not $E_{Y+1}S_X$. One could manage this situation by having a smart index that remembers which file to read to get the state for a particular superstep. However, this would then require an extra read IO, increasing the total amount of read IOs to 2: one to read $E_{Y+1}S_X$ containing the state and one to read $E_{Y+2}S_X$ containing the deltas.

Fortunately, a simple change to the read/write pattern allows the use of the same number of total IOs even when machines are not involved in intermediate events. This is achieved by replacing the 1 read/2 writes with 2 reads/1 write and keeping an index of existing files. Consider the execution of superstep S_X by 2 events E_{Y-C} and E_Y which represent 2 consecutive events in which a specific machine was involved. If $C > 1$ then that machine was not involved in the events in between. When E_{Y-C} finishes execution of S_X, it will write the updated state and new deltas to a single file $E_{Y-C}S_X$. When E_Y starts executing S_X, it will do 2 reads:

- State will be read from the file $E_{Y-C}S_Z$ where S_Z represents the last superstep (with $0 \leqslant Z \leqslant X$) in which E_{Y-C} actually executed and wrote updated state. If $Z = X$ then the state can be read directly without any modification as it pertains to the same superstep. If, however, $Z < X$ then the read state needs to be transformed to become a feasible state for superstep S_X. This transformation entails setting all *valueIn* of the new state to the old *valueOuts*, setting empty *valueOut* and emptying the mailboxes. Finally, if there is no preceding event or the machine only got involved in previous events after S_X then an empty state is used instead.

- Deltas will be read from the file E_YS_Z where S_Z represents the last superstep (with $0 \leqslant Z \leqslant X$) in which E_Y actually executed and wrote updated deltas. If this is the first superstep in which this machine got involved in event E_Y then an empty delta is used instead.

Figure 14 exemplifies the application of these read strategies in a graphical manner.

This new mechanism also brings with it some extra benefits:

- It reduces the time spent waiting for IO operations to complete since read operations are usually faster than write operations in storage media.

- It allows the existence of gaps in the memoized deltas for a particular event. The previous IO access scheme relied on doing a single read operation and therefore needed all the information for executing a superstep to be in the same file. Since deltas were only written to the file of the next superstep, this meant that even in those supersteps where a machine did not have anything to compute, it still needed to read the deltas from the current superstep file and write it to the next superstep file.

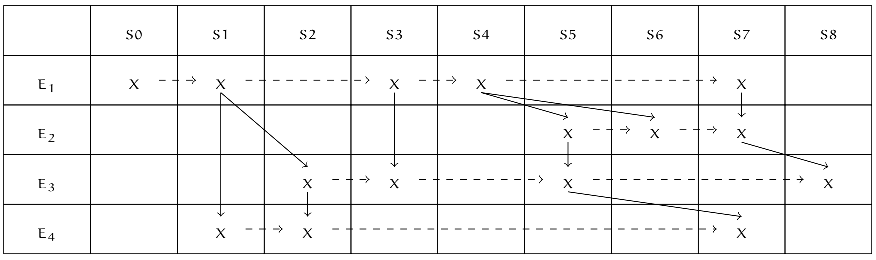


Figure 14: Example of the new read pattern registered during the execution of 4 events for 9 supersteps. X represents positions where state and delta information resulting from an actual execution of a superstep by an event was written. Solid arrows represent state reads while dashed arrows represent delta reads. Diagonal arrows represent state reads where a transformation has to be applied.

Consider a situation where event E_Y runs for 9 supersteps before completing but a specific machine only has to actually do any computation in 2 of those supersteps (say S_1 and S_8 for example). With the previous scheme, enforcing the single read IO meant needing to propagate the deltas generated in S_1 to all the intermediate files $S_2, S_3, \dots, S_8$ so that, when executing S_8, the deltas generated originally in S_1 would be directly accessible. With the new scheme, when trying to execute S_8, one simply has to check the index and obtain the address of the files containing the state ($E_{Y-1}S_8$) and the deltas (E_YS_1).

- It allows the existence of gaps in the memoized state for a particular event. With the previous IO access scheme one had to ensure that all superstep files had a correct state for all supersteps of an event. Since superstep files were shared between events this meant that the first event to create a new file while writing the delta for the next superstep also had to construct a new state from its current one and write it to the file as well. Having different files per event, as happens with the new scheme, would apparently require writing the correct state in every superstep even if it was one where nothing changed. In this way the subsequent event would always be able to find the right state for S_X by looking in $E_{Y-1}S_X$. However, this is not needed, when using the state reading strategy described above.

5 LOCAL SYNCHRONIZATION

5.1 INTRODUCTION

Giraph and RTGiraph both follow the Pregel model described in Section 2.1. In this model, computation is divided in supersteps with a node advancing to the next superstep if and only if all other nodes have already finished the current one. This simple synchronization mechanism, also called global synchronization, is easy to implement and, in ideal conditions where all the nodes are equally balanced, is very effective. Unfortunately, with real-world data and cluster deployments, it is virtually impossible to obtain a perfectly balanced distribution of the computational load across a set of servers. Imperfect input partitioning, network jitter and non-deterministic hardware performance variations are but a few of the factors that contribute to this phenomenon. Under the presence of the aforementioned imbalances, global synchronization mechanisms experience great impacts to the execution performance due to the appearance of stragglers. Stragglers are nodes that take a greater amount of time to finish an execution compared to the execution times of their peers. Since, with global synchronization, each node has to wait for all other nodes to complete the current step before continuing, global performance is capped by the performance of these stragglers and resources of non-stragglers become underutilized as seen in Figure 3.

To solve this issue, some systems [16] [12] remove all synchronization barriers and allow each node to compute at its own pace. While this asynchronous execution effectively solves the straggler issue by removing the requirement that each node wait for its peers, it introduces some disadvantages such as the requirement for specially crafted algorithms that are asynchronous in their nature and the increased complexity of coordination among nodes. This last point is of particular importance as it suggests that while asynchronous execution keeps nodes from becoming idle waiting for others, it may do so at the expense of unnecessary computations and message sending.

Given the issues identified with the two extremes of the synchronization spectrum, a natural follow-up is to wonder whether a combination of the 2 mechanisms could provide an increase in performance in the presence of imbalances while allowing the execution of traditional synchronous algorithms. This mechanism, called local synchronization, replaces global cluster-wide barriers with local barriers among neighborhood dependencies. To support this, however, the communication assumptions currently employed by these systems had to be revised.

5.2 COMMUNICATION ASSUMPTIONS

In the Pregel computational model on which Giraph is based, no restrictions are made related to communication patterns between different vertices. A vertex may send a message to any other vertex in the graph independently of the existence of an edge connecting the two vertices.

Supporting this communication pattern would mean that a vertex may receive a message from any other vertex in a certain superstep and, as such, would need to wait until all other vertices computed before moving on with its own computation. This would defeat the purpose of any local synchronization implementation as the local barriers would in fact manifest themselves as a single global barrier (as currently happens).

For local synchronization to be able to improve the performance of the execution of applications and events over big graphs, stricter communication patterns have to be enforced. In particular, enforcing all inter-vertex communication to occur over graph edges allows us to obtain information about the computational dependencies by analyzing the edges that connect separate computational units of the graph. Since most graph algorithms already follow this assumption, its adoption may only cause incompatibilities with a small fraction of applications.

5.3 DESIGN AND IMPLEMENTATION

5.3.1 *Partition-level*

In Giraph and RTGiraph, the graph is not treated as a single computational unit. Instead, each node is responsible for managing multiple partitions of the graph. In doing so, scalable distribution of the processing of the graph data is enabled.

Being the computational unit located at the highest abstraction layer at the level of a node (as shown in Figure 15), partitions are a natural starting point for experimenting with local synchronization.

Each partition represents a subset of the graph and, as such, contains a subset of the total vertices in the graph along with their connections. These connections can link two vertices local to the partition (a local connection) or can connect a local vertex to an external one located in a different partition (an external connection). By keeping track of these external connections between different partitions, a graph of inter-partition dependencies can be identified. This graph constitutes the partition meta-graph shown in Figure 16 for a 20-node graph.

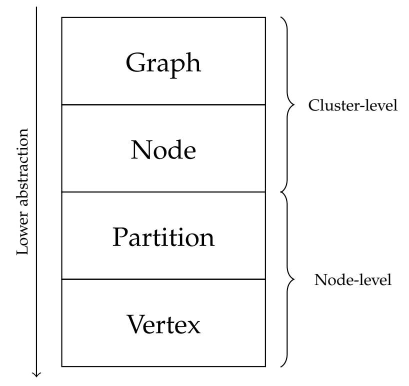


Figure 15: Computational abstraction layers.

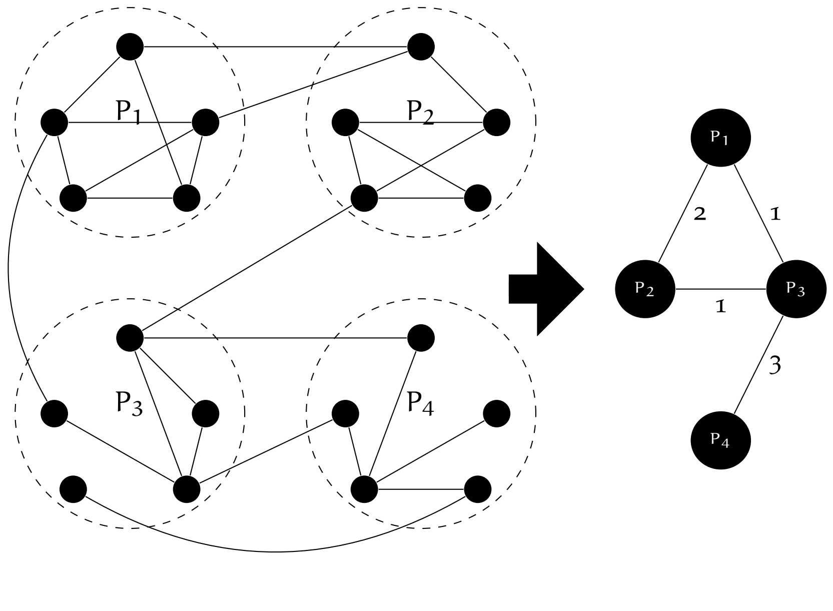


Figure 16: A 20-node graph with four partitions (on the left) and the corresponding partition meta-graph (on the right). The edges on the partition meta-graph are labeled with the number of edges on the vertex graph that compose that connection.

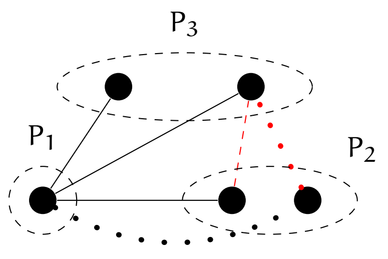
Fortunately, the separation of responsibilities in several components which underlies the design of RTGiraph allowed for a non-invasive implementation of partition-level local synchronization requiring modifications to the storage of graph partitions and the creation of a new type of event executor. These changes are described in the following sections.

5.3.1.1 *Graph partitions*

Graph partitions in RTGiraph contain information about the vertices that compose the partition as well as the mutations made to these vertices (or their connections) by each event. Since these objects are responsible for holding all information pertaining to the actual structure of the graph, they are prime candidates for holding information about the partition meta-graph. To that end, 2 new sets were added to each graph partition object: a set of all the partitions that have outgoing edges with the current partition as the destination (incoming partition set); and a set of all the partitions to which the current partition has outgoing edges (outgoing partition set).

Considering that, at any one time, RTGiraph might be processing a multitude of events, these two sets have to support multiple views of the existing connections depending on the requesting event. To achieve this, a complete connection set is generated for the base graph structure including the number of edges involved in that connection. Afterwards, for each event, only deltas of the number of edges connecting to a certain partition are stored. When a certain event, say E_2, asks for the incoming connections of a partition, an initial set is created from the complete base connection set. Afterwards, deltas for events E_1 and E_2 are applied by adding their values to the edge counts of the affected partitions. When all deltas have been applied, those connections which are now associated with an edge count of 0 are removed from the set and the final incoming connection set for that partition in event E_2 is obtained. Figure 17 exemplifies the behavior described in this paragraph.

When dealing with a large amount of events, having to reconstruct each connection set from a series of deltas starting with the first event is bound to become prohibitively expensive. To counter this, deltas can be periodically compacted (along with the mutation information also stored in the graph partition), turning the last completed event into the new base initial event. To do so, one just has to follow the algorithm described in the previous paragraph to obtain the connection view for the aforementioned event; set it as the new base connection set; and remove all the deltas for events with equal or lower identifier.



	P_1	P_3
E_0	1	2
ΔE_1		-1
ΔE_2	$+1$	-1
	$\downarrow$	
E_2	2	0
Conn. Set	$\{P_1\}$	

Figure 17: Example of the delta-based partition meta-graph store. The table on the right represents partition meta-graph information stored by partition P_2 for the base event E_0 and 2 new events: E_1 and E_2. Dashed edges on the graph represent edges affected by event E_1 while dotted edges represent edges affected by event E_2. Red edges show edges that were removed by the event affecting them.

5.3.1.2 *Local sync event executor*

It is through the event executor that the actual node-level partition execution scheduling is done. As described in Section 3.3.6, an event executor essentially follows the same steps for every superstep:

1. Reach start barrier and wait for peers.
2. Launch partitions that need executing in the current superstep and wait for their completion.
3. Send outgoing messages generated during execution.
4. Reach end barrier and wait for peers.

While this is compatible with a global synchronization mechanism, for local synchronization the launching of partitions cannot wait until the breaking of the end barrier of the previous superstep and the start barrier of the new superstep. Neither can the sending of messages be delayed until all partitions have finished execution, otherwise remote nodes would only be notified of the completion of partitions when the entire node responsible for them finished the superstep. However, the global synchronization module already contains all the logic for determining when an event can start or has finished execution so reusing the global start and end barriers would enable us to make use of this already implemented functionality.

Taking into account the issues described in the previous paragraph, the implementation of the local sync event executor is based on 2 parallel progress layers:

1. Event execution progress layer.

2. Partition execution progress layer.

The first layer is responsible for advancing and keeping track of the progress of the event as a whole. Its main purpose is to communicate with the global synchronization module by reaching the start and end barriers as if this were an execution being done by the non-local sync event executor. However, this layer no longer directly manages partitions or their messages.

The second layer is responsible uniquely for the management of partitions, message sending and dependency tracking. It analyzes messages received from other nodes in addition to local notifications to determine the set of partitions that have finished in each active superstep. When a partition tries to move on to the next superstep, this set of finished partitions, along with the incoming connection set of the partition, are consulted to decide whether it can indeed proceed to the next superstep or if it has to wait until more of its dependencies finish execution. Whereas, before, all partitions worked in unison as part of the global execution, now, each partition is treated as an independent component, executing at the rate set by its dependencies.

Another difference from the previous executor model is that, while before each remote delta was associated with at least one message sent during the computation of a vertex in the source partition, with the new model remote deltas might be empty. These empty remote deltas are used solely for dependency tracking and are sent to indicate the end of execution of a partition on a certain superstep when said execution did not generate any algorithmic messages to the destination partition. Therefore, under the new model, when a partition finishes execution a remote delta is guaranteed to be sent through each outgoing connection of that partition independently of the existence or not of actual messages to send.

Figure 18 and Figure 19 describe the basic algorithm adopted by each of the aforementioned layers through the use of flowcharts. While the flowcharts are straightforward and similar to the one shown in Figure 9, the initial steps in the event layer flowchart highlight a quirk with the implementation of local synchronization. Because a partition will only advance when it is sure that all of its dependencies have completed, these dependencies have to somehow notify the former of their completion. For the initial base event, this is not an issue as all partitions are involved with the event from the start. However, while processing the following events, only the partitions directly involved in the mutation are initially involved in the event. Those partitions, however, still expect notifications from their dependencies which might not even be aware of the existence of this new event. To address this issue, upon the first execution of a partition, its dependencies are made aware of the existence of an event. In the case of undirected graphs this is done automatically when sending messages through the outgoing links. A node that is not involved in an

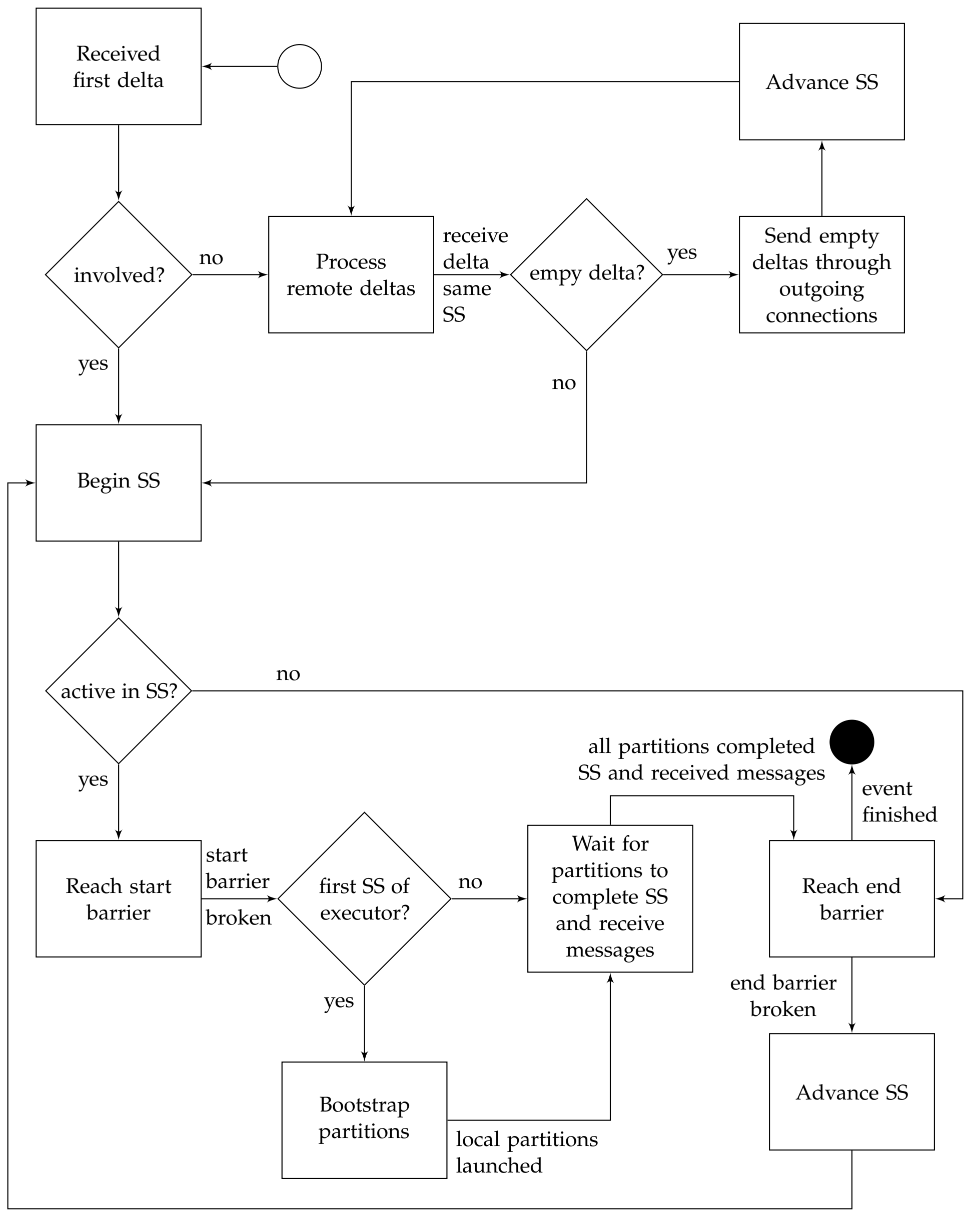


Figure 18: Flowchart of the partition-level local sync event layer. Blank circle symbolizes start of flow, filled circle symbolizes end of flow.

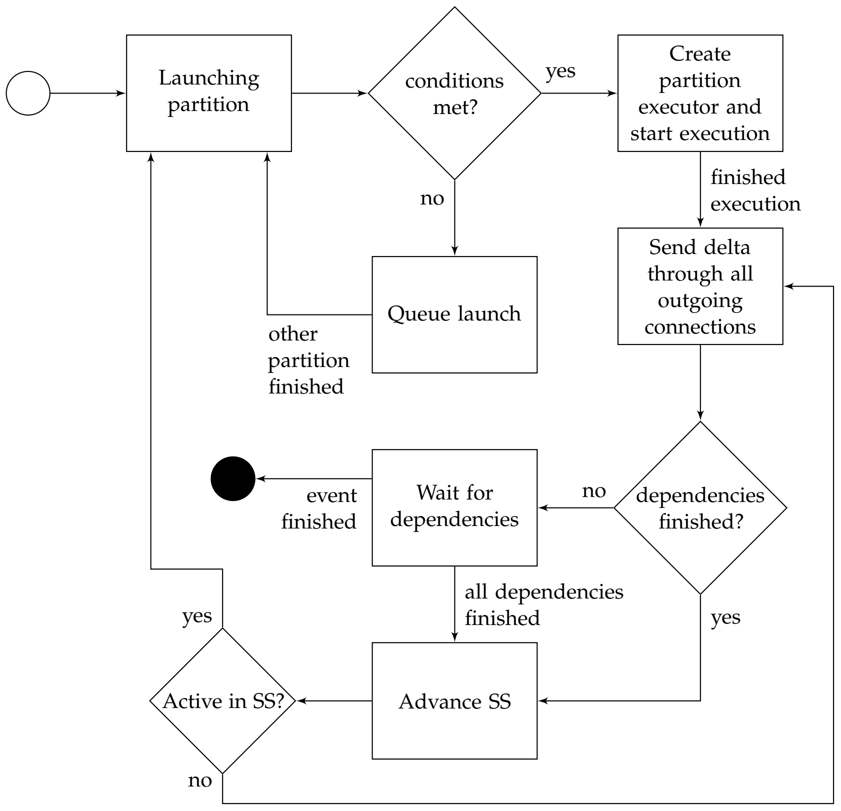


Figure 19: Flowchart of the partition-level local sync partition layer. This is the execution flow followed by every partition. Blank circle symbolizes start of flow, filled circle symbolizes end of flow.

event will simply send empty remote deltas to all outgoing partitions, notifying them of their execution completion (or, in this case, absence of execution) for that superstep. When a node that was not previously involved receives a non-empty remote delta (that is, a delta containing an actual algorithmic message to be handled by local vertices), the node becomes involved in the event at the superstep of the received delta: it starts participating in the reaching and breaking of barriers and bootstraps local partitions for execution.

In the partition-layer flowchart, one can see mention of *launch conditions*. Before, the single launch condition was that the number of active partitions be below the threshold defined in the configuration. If the number of active partitions were equal to the threshold, the partition whose launch had just been attempted would be queued until one of the running partitions finished execution. With localsync, the set of conditions changes. Before launching partition P_Z for event E_Y and superstep $S_{E_Y P_Z}$, all the following conditions must be met:

1. $activePartitions < maxActivePartitions$ (as before).
2. $S_{E_Y P_Z} - S_{E_Y} \leqslant maxSAhead$ — if the superstep of the partition about to be launched is more than $maxSAhead$ supersteps ahead of the oldest still active superstep (by definition that of the event layer) the launch should be queued. This condition is important for 2 reasons:
   - It offers a way of limiting memoized state in the presence of independent connected components in the partition meta-graph. In the presence of these independent components, the non-existence of dependencies allows components to run at their own pace. If the components are imbalanced, the faster to compute could be running hundreds of supersteps ahead of the slowest. Since, for each superstep computed ahead, one has to keep some extra state in memory, having a mechanism to limit the size of this state is essential for memory management.
   - The end of an event is determined by the global synchronization module. As can be seen in Figure 19, the partition execution flow is totally independent of the global synchronization module, only caring about dependencies. Therefore, until the global synchronization module reports the end of the event to the event executor and the event layer stops listening and dispatching events, the partition layer will continue advancing the superstep of sleeping partitions well beyond the real termination superstep of the event. By limiting the maximum number of supersteps a partition can be ahead of the main event flow, we limit this number of unnecessary advancements at the end of an event execution.

3. $S_{E_{Y-1}P_Z} > S_{E_Y P_Z}$ or E_{Y-1} has finished. — Because the partition execution layer operates independently of the global synchronization module, it can no longer rely on the later to ensure correct scheduling when running in an event pipeline (see Section 4.2.1). To address this, a static manager accessible by all event executor threads local to a node was created to keep track of the last superstep finished by local partitions between different events. Using this manager, determining the truth value of this condition becomes trivial.

5.3.2 *Vertex-level*

The previous section described how a local synchronization system was implemented to attempt to relax synchronization and stop relying on global barriers while continuing to ensure correctness by ensuring that each partition only proceed when its dependencies have finished. In doing so, one ensures that each such partition will execute with the exact same information as in a globally synchronous execution.

Looking back at Figure 15, there is yet another abstraction layer that may be explored to attempt to address this issue: the vertex-layer. From the partition viewpoint one might identify 2 types of vertices:

- Local vertices — Vertices whose neighbors are also contained in the partition. By definition, when a partition has finished computing a superstep, all the dependencies of these vertices have also finished computing so they can be immediately executed.
- Frontier vertices — Vertices which have one or more neighbors contained in different partitions. These vertices must wait until receiving notification of completion from the neighboring partitions before being able to be computed for the next superstep.

Assuming that the used partitioning scheme is smart enough to try to maximize the number of local vertices, exploiting these local vertices might allow the computation of a sizable subset of the partition even before any message is received from other partitions. It is also likely that frontier vertices only share a small subset of the dependencies of the whole partition and, as such, could be computed before the entirety of the partition dependencies have finished.

One might consider using a system similar to that employed in partition-level local synchronization to execute vertices as soon as all their dependencies have been met. However, the overhead introduced by requesting hundreds of executions of tiny subsets of a partition is bound to be prohibitive as vertex information is stored in a serialized memory-friendly manner.

Instead, the chosen implementation of vertex-level local synchronization reuses the partition dependency tracking mechanism of partition-level local synchronization but limits the number of execution passes over a partition in the same superstep to 2:

1. First pass — During the first pass, all local vertices and external vertices whose dependencies have already been met will be executed.

2. Second pass — During the second pass, all vertices not executed in the first pass are executed. Therefore, at the end of the second pass, the entire partition will have been executed.

Referring back to the description of the execution passes, it becomes clear that the second pass may only occur when all partition dependencies have been met because a second pass is guaranteed to also be a final pass (a pass in which the entire partition is computed). What is not evident, however, is at what time a first pass should be attempted: too early implies that a very small number of vertices is computed; too late and the second pass will be the one computing a very small number of vertices. Ideally each pass would execute roughly 50% of the vertices. In the implementation of the system, 2 configurable options were included to allow tuning of the two-pass system to different applications and inputs:

- *rtgiraph.local.sync.vertex.min* — Minimum percentage of dependent partitions finished before attempting a first pass.

- *rtgiraph.local.sync.vertex.max* — Maximum percentage of dependent partitions finished after which the first pass is treated as a final pass.

It is interesting to note that some supersteps might not even have two passes: if all dependencies finish (or a percentage bigger than the one specified by *rtgiraph.local.sync.vertex.max*) before a first pass is attempted, then that partition will be executed in a single pass for that superstep, thus taking advantage of the speed up in global execution.

Implementing this vertex-level local synchronization required the creation of a new event executor based on the one developed for partition-level local synchronization, as well as a modification of the partition executors to support multiple passes and the merging of generated state and deltas for those different passes. These changes will be described in the subsections to follow.

5.3.2.1 *Vertex local sync event executor*

The new event executor for vertex-level local synchronization reuses the dependency tracking mechanism of partition-level local synchronization while adding a set of hooks to the execution flow that handle the two execution passes logic. Figure 20 highlights the main changes to the partition execution layer. Event execution layer remains unaltered.

To be able to run an incomplete pass, all following requirements must be met:

1. Partition is not running nor queued for launch.
2. Percentage of finished dependencies must be between the configurable values *rtgiraph.local.sync.vertex.min/max*.
3. Partition must not have done an incomplete pass in the same superstep.
4. All requirements for launching a partition as described at the end of Section 5.3.1.2.

When an incomplete execution pass over the partition is done, a reference is kept to the partition executor used so that it (and its stored state) can be reused in the final execution pass. As soon as a final pass is done, the resulting data is saved and the executor is destroyed.

5.3.2.2 *Vertex local sync partition executor*

Previous partition executors were run only once and, as such, did not store any intermediate state and were discarded as soon as the execution of the partition finished. To support a two-pass architecture, new partition executors need to keep track of the current execution state through the following structures:

- *alreadyComputedVertices* — Set of vertices that have been executed on a previous pass.
- *nextState* — Bytearray to which updated state is written during execution of a pass over the vertices of the partition.
- *previousPassNextState* — Bytearray to which updated state was written during execution of the previous pass (if applicable).
- *nextDelta* — Bytearray to which updated deltas are written during execution of a pass over the vertices of the partition.
- *previousPassNextDelta* — Bytearray to which the updated deltas were written during the execution of the previous pass (if applicable).

Because state and deltas are stored in bytearrays for efficient memory storage and because these bytearrays need to be ordered by vertex identifier for the system to work correctly, the results of each pass cannot be written directly to a single final structure shared by the

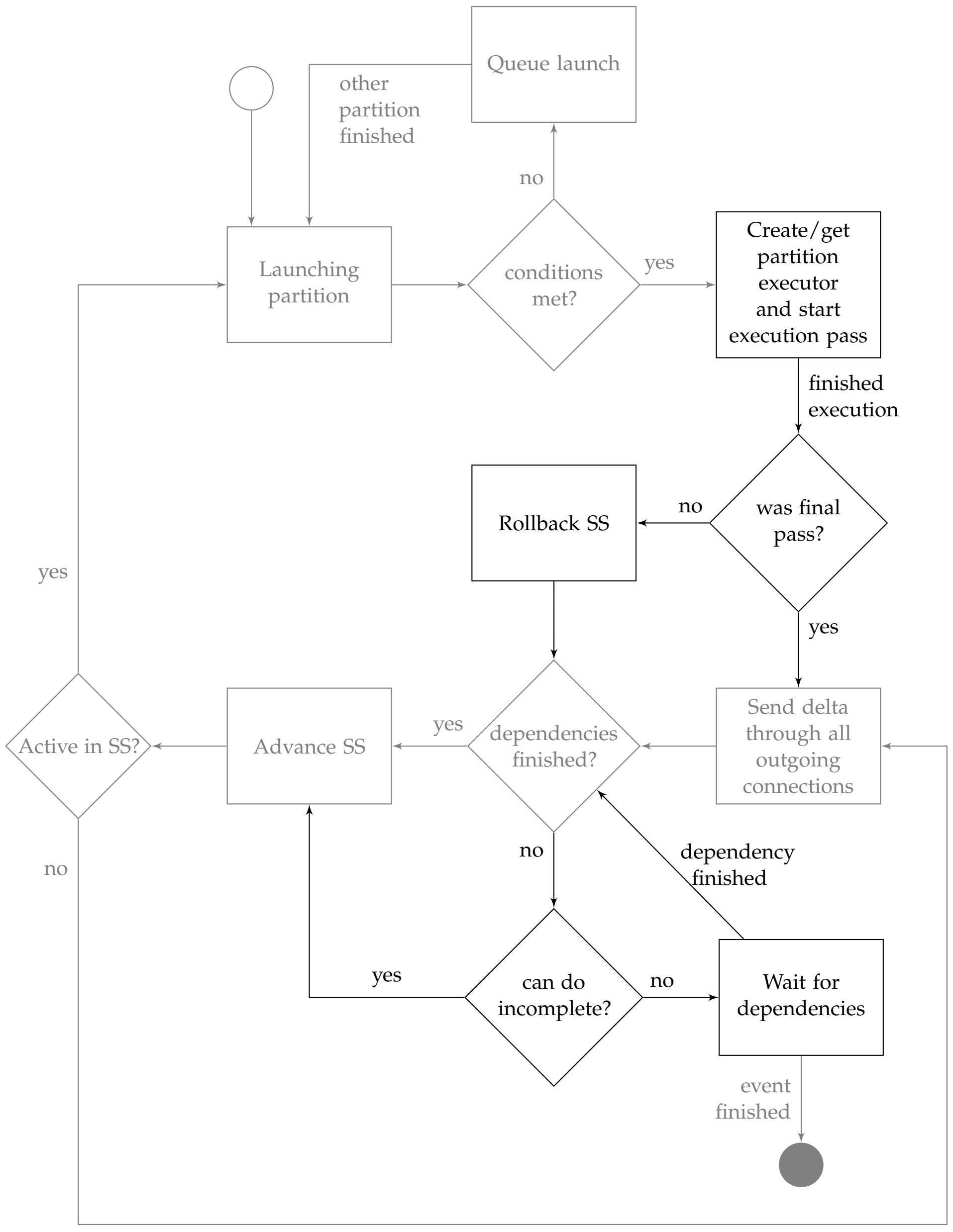


Figure 20: Flowchart of the vertex-level local sync partition layer. This is the execution flow followed by every partition. Grayed out elements represent functionality inherited from partition-level local sync while highlighted elements represent modifications. Blank circle symbolizes start of flow, filled circle symbolizes end of flow.

two passes. Instead, each pass writes its updated state and deltas, in order, to separate bytearrays. After the final pass finishes execution, the 2 bytearrays for the state and for the delta are merged using a 2-way merge mechanism executed in linear time with the size of both arrays. If a superstep is executed in just a single final pass then this is not needed and the old behavior suffices.

For each execution pass of the partition, an updated set of the unfinished dependencies of that partition is provided. If this set is empty, all dependencies of the partition have finished so the current pass is set as a final pass. Otherwise, only a fraction of the dependencies have completed and some vertices might have to be skipped for computation in the next pass.

When iterating over the vertices of the partition, only those vertices that satisfy the following conditions can be computed:

- Vertex is not present in *alreadyComputedVertices*.
- Vertex dependency set has to be disjoint with the set of unfinished partitions.

If a vertex satisfies the conditions, it is computed (if needed) and the result of the computation is stored on the state and delta objects. If, on the contrary, a vertex violates any of the conditions, it is ignored while doing the current pass and its execution is left for the following one.

After finishing a non-final pass, the event executor that created this partition executor is notified but the new state and delta are not passed as arguments nor updated on-disk as they are still incomplete. Only after the execution of a final pass are these structures committed to disk and made available to the remaining components of the system.

6 EVALUATION

6.1 INTRODUCTION

Having reasoned about possible mechanisms to take better advantage of idle time and having implemented them as described in the previous chapters, one now has to understand how these techniques fare out of the theoretical realm. To that end, the new implemented modules were submitted to executions over both specially tailored graphs (highlighting favorable scenarios of execution) and a real-world graph collected from the Spanish social network Tuenti. The details of the execution environment, datasets, algorithms and experiments done will be detailed over the next few sections.

6.2 EVALUATION ENVIRONMENT

6.2.1 *Hardware*

All experiments detailed in this chapter were run on Amazon EC2 clusters composed of up to 21 memory-optimized *r3.xlarge* instances with the following configuration [2]:

- 4 vCPUs running over Intel Xeon E5-2680 v2 (Ivy Bridge) processors.
- 30.5 GiB of RAM.
- 8 GB of EBS-backed root storage.
- 80 GB of SSD storage.
- High performance networking.

6.2.2 *Software*

Each cluster was configured using a specially crafted Fabric[1] script that deployed and configured the following software in the cluster:

- Oracle Java SE Development Kit 7u55.
- Apache Hadoop 2.4.0.
- Apache ZooKeeper 3.4.6.

1 Fabric — a Python framework for streamlining application deployment and system administration (fabfile.org)

- Cloudera 5 Apache Pig 0.12.
- YourKit Java Profiler 2014 EAP, build 14050.

6.2.3 *Configuration*

Of the set of machines composing the cluster, 1 machine was chosen as the master, exclusively running the coordination daemons required for execution, namely, Hadoop YARN's resource manager, Hadoop HDFS' namenode and a ZooKeeper instance. The remaining machines were setup as slaves, running the daemons responsible for doing actual work: Hadoop YARN's node managers and Hadoop HDFS' datanodes.

Machines were configured to make use of the private network interfaces when doing intra-cluster communication, thus exploiting the high speed local network.

Hadoop daemons were configured to place all data (temporary or not) in the high speed SSD ephemeral storage. All vCPUs of each slave machine were made available to running Hadoop MapReduce jobs as well as approximately 28GB of the total RAM. MapReduce jobs were setup to use new YARN computational framework thus taking advantage of all the resource management improvements in MapReduce 2 [32]. Each node was also configured to run at most 1 mapper at a time, thus ensuring that it hosted a single instance of RTGiraph.

ZooKeeper was kept with the default configurations except for the data storage path which was also placed in the ephemeral SSD disk.

Regarding the actual configuration of RTGiraph, every execution in this evaluation section uses the following configuration as base:

- Pull messaging.
- Incremental computation.
- Graph caching in memory.
- State and delta caching in memory.

6.3 datasets

6.3.1 *Artificial Graphs*

In order to exert a greater control over the graph structure and to explore a variety of situations favorable to the new mechanisms, a set of artificially generated graphs was used in some of the experiments. The actual graphs used depend on the type of experiment and, as such, are described in more detail when considering their individual accounts.

	Min	Max	Avg	Total
Vertices	52,960	67,551	61,828	3,709,730
Local Vertices	410	1042	650	39,007
Edges	3,907,864	3,947,309	3,926,644	235,598,676

Figure 21: Tuenti graph statistics. Minimum, maximum and average values are per partition. Total represents the aggregate over all partitions.

All artificial graphs were generated by custom-made Python scripts leveraging NetworkX[2].

6.3.2 *Tuenti*

The real-world graph used in the experiments in this chapter corresponds to a snapshot of the activity in the Spanish social network Tuenti. Each node of the graph represents a user of the social network and edges represent friendship relations between these users.

This graph was partitioned into 60 different partitions using the Spinner [18] partitioning algorithm. With a total of 20 slave nodes in the cluster, the partition to node assignment was made in circular order, totaling 3 partitions per node.

Figure 21 summarizes the main statistics of this graph.

6.4 APPLICATIONS

6.4.1 *Dummy*

Just as artificially generated graphs provide a fine-grained control of the graph structure, the dummy applications described in this section allow a finer-grained control of the execution flow. These dummy applications are used in tandem with artificially generated graphs to showcase particular scenarios of relevance to some of the experiments.

6.4.1.1 *Dummy Skewed*

The Dummy Skewed application uses a deterministic vertex-to-partition identifier resolution scheme (based on the vertex identifier attribution logic used when generating the artificial graphs) to identify the partition to which a vertex belongs. Then, based on the vertex's partition, it simulates a variable computational load by varying the number of execution steps taken during the computation of that vertex.

2 NetworkX — A Python library for graph management. (networkx.github.io)

Algorithm 1: Per-vertex algorithm for the Dummy Skewed Application

```
input : vertex, superstep, maxSuperstep
if superstep == 0 then
    vertex.value = 0;
numSteps = getNumberSteps();
for i = 0 to numSteps do
    vertex.value += random();
if superstep == maxSuperstep then
    vertex.halt();
```

The actual algorithm used by this application is detailed in Algorithm 1. Each vertex executes the specified algorithm for 5 supersteps before halting. The variation in the computational load described in the previous paragraph is introduced by varying the upper limit of the *for* cycle:

- Vertices in partition 1 always perform the same amount of steps in the *for* cycle: 200.
- Vertices in partition x with $x > 1$ perform an amount of steps according to Equation 1. Thus, each partition performs roughly twice the computation of its predecessor.

$$numSteps = 2^x * 100 \tag{1}$$

This artificial skewness attempts to simulate a situation where, due to incorrect partitioning, node problems, or application-specific issues; some partitions consistently require bigger slices of computation time than others.

6.4.1.2 *Dummy Skewed Alternating*

The Dummy Skewed Alternating application is an extended version of Dummy Skewed where partitions other than partition 1 perform a varying amount of computation per superstep. In particular, Equation 1 is replaced by Equation 2.

$$numSteps = 2^{[(x+superstep)\%numPartitions+1]} * 100 \tag{2}$$

With this change, the imbalance of the computational load cycles through all partitions other than partition 1. Therefore with a total of 4 partitions, a partition whose vertices, in a previous superstep executed 16000 iterations of the cycle, in the next superstep would execute only 4000, followed by 8000 on the next superstep and then back to 16000.

This change in artificial skewness attempts to simulate a situation where, due to the same reasons described at the end of Section 6.4.1.1, some partitions require bigger slices of computation time than others but at different steps of the application execution.

6.4.2 *Real-world*

In this section we describe the real-world applications used in our experiments with the Tuenti graph.

6.4.2.1 *Single Source Shortest Paths*

Single source shortest paths (or SSSP for brevity) is an application that calculates the length of the shortest path from every vertex in the graph to a designated source vertex.

Vertices in SSSP store as a value a single double containing the lowest known distance to the source. To find the correct value for each vertex, SSSP starts a Breadth First Search (BFS) by setting the value of the source to 0 and then triggering the sending of a message to all of its neighbors with value 1. All other nodes are initialized with a value of infinite. When a vertex receives a message, that represents the existence of a path from the source to that vertex with distance equal to the value in the message. Therefore, upon receipt of a message, a vertex compares its value to the received one and, if the former is bigger than the latter, updates its vertex value and sends messages through its outgoing links with the new value plus 1. Algorithm 2 contains the pseudocode of the algorithm executed by each vertex.

Algorithm 2: Per-vertex algorithm for Single Source Shortest Paths

```
input : vertex, messages, superstep
if superstep == 0 then
    vertex.value = Inf;
minDistance = isSource(vertex) ? 0 : Inf;
for message ∈ messages do
    minDistance = min(minDistance, message.value);
if minDistance < vertex.value then
    vertex.value = minDistance;
    sendMessageToNeighbours(vertex, minDistance + 1);
vertex.halt();
```

Because this algorithm is equivalent to a BFS, the first time a vertex receives a message and updates its value, it is guaranteed to be for the shortest path and, thus, the value set is final. As a result, each vertex will send messages to its neighbors at most once during the execution of the application. In addition, because vertices always halt at the end of a superstep in which they executed and are only reactivated with the receipt of a new message in a later superstep, the number of executions done by a vertex is bounded by the number of neighbors it has. These properties, along with the simplicity of the algorithm itself, make SSSP a very lightweight application both in terms of messages and computations. Because both vertices and messages use a double (8 bytes) as their value, the memory overhead of state and delta storage is also very low. In addition, SSSP is very prone to imbalances as it is very likely that the partitions closer to the source perform the biggest amount of computations at the beginning of the execution and the least amount towards the end.

6.4.2.2 *PageRank*

Pagerank is an algorithm best known for being the first algorithm used by Google to order search results. This algorithm assigns numerical values to each vertex in the graph, representing the relative importance of the vertex compared to the remaining ones. Given a random walk through a graph, these values aim to represent the likelihood that a vertex will be the last vertex in the walk.

The original PageRank algorithm already relied on the concept of multiple iterations or passes over each vertex and, as such, transfers directly to the Pregel model. Initially, all vertices are set to the same value $V = 1/N$ (where N is the total number of vertices) and send, through each outgoing edge, a message with the value $V/outDegree$. In the following supersteps, vertices will sum the values of received messages and update their internal value V according to Equation 3 (with M representing the total set of received messages and m representing individual ones).

$$V = \frac{1-d}{N} + d * \sum_{m \in M} m.value \tag{3}$$

In this equation, d represents a damping factor which constitutes the probability that, at any step, the current random walk will terminate and another one be started at a random vertex. A value of $d = 0.85$ is often used and is also the one used in these evaluations. A vertex halts as soon as it reaches the maximum defined number of iterations. This behavior is described in Algorithm 3

For the experiments described in the remainder of this chapter, the value of *maxSuperstep* used was 3. Unlike SSSP, in PageRank, all the vertices of the graph are involved in the execution until the maximum number of supersteps is reached. Furthermore, each vertex execution triggers the sending of messages to all neighbors. Therefore, PageRank maximizes the number of computations and messages sent at each superstep. However, each computation itself is still very simple, requiring only a sum over the messages of all received messages followed by a sum of a quotient with a product. Vertex and

Algorithm 3: Per-vertex algorithm for Pagerank

```
input  : vertex, messages, superstep, numTotalVertices
if superstep == 0 then
    vertex.value = 1/numTotalVertices;
else
    sum = 0;
    for message ∈ messages do
        sum += message.value;
    vertex.value = (0.15/numTotalVertices) + 0.85 * sum;
if superstep < maxSuperstep then
    sendMessageToNeighbours(vertex,
    vertex.value/vertex.outdegree);
else
    vertex.halt();
```

message values are also lightweight, each consisting of a *double* variable (8 bytes). Taking these facts into account, the impact in CPU and memory should be higher than that of SSSP but still manageable. Moreover, as all vertices are involved in each superstep, PageRank should be less susceptible to imbalances caused by application progress.

6.4.2.3 *Semi Clustering*

Semi Clustering is a clustering algorithm that groups vertices into clusters while allowing each vertex to belong to more than one cluster. The version used in these experiments corresponds to the one presented in [17]. The most common use case for Semi Clustering are social graphs where one might want to identify groups of people who interact frequently with one another at the expense of interactions with people outside the group. The weight of the edges connecting two vertices represents the strength or frequency of interaction.

Algorithm 4 presents the algorithm for Semi Clustering in pseudocode. 2 parameters determine the execution of the application:

- C_{max} — Maximum number of semi-clusters to identify.
- V_{max} — Maximum capacity (number of vertices) of a semi-cluster.

The value of each vertex is a list with at most C_{max} semi-clusters. Initially this list is populated with a single semi-cluster containing exclusively the vertex that owns the list. This cluster is also given the initial score of 1. At the end of a superstep the current semi-cluster list of the vertex is sent to its neighbors. In the following supersteps, each vertex iterates over the received semi-clusters and adds itself to

Algorithm 4: Per-vertex algorithm for Semi Clustering

```
input  : vertex, messages, superstep
if superstep == 0 then
    vertex.value = [semiCluster(vertex.id)];
    sendMessageToNeighbours(vertex, vertex.value);
else if superstep > 0 and superstep ≠ maxSuperstep then
    candidates = [];
    for semiCluster ∈ messages do
        candidates += semiCluster;
        if not semiCluster.contains(vertex.id) then
            if not semiCluster.full() then
                semiCluster += vertex.id;
    candidates.sort();
    sendMessageToNeighbours(vertex, candidates.top($C_{max}$));
    vertex.value = candidates.top($C_{max}$).containing(vertex.id);
vertex.halt();
```

those semi-clusters that are not full. Following that, scores for the modified semi-clusters are recalculated according to Equation 4

$$\text{Score} = \frac{I - f_B * B}{V(V-1)/2} \tag{4}$$

where I and B correspond, respectively, to the sum of weights of internal and boundary edges. V represents the current size of the semi-cluster and f_B is a user-specified parameter between 0 and 1. Finally, the vertex sends the truncated cluster list with C_{max} elements to its neighbors and retains the C_{max} semi-clusters with highest score that contain it as its new value . This algorithm runs until reaching the user-specified superstep limit which was set to 5 in the experiments here described.

Like SSSP, vertices in Semi Clustering halt as soon as they finish computation of a superstep. However, because in each such computation they send messages to their neighbors (not just on the first computation), vertices located in connected components end up computing in every superstep like what occurs with PageRank. As opposed to both SSSP and PageRank, however, Semi Clustering's per-vertex computation is relatively heavy, creating, appending, sorting and truncating lists, as well as recalculating semi-cluster scores. In terms of memory, each vertex and message holds at most C_{max} semi-clusters with V_{max} vertex ids (4 byte integers) each. Each semi-cluster also stores 3 doubles (8*3 bytes): *innerScore*, *boundaryScore*, *totalScore*. When serialized these semi-cluster lists take a maximum size of $C_{max} * ((1 + V_{max}) * 4 + 3 * 8) + 1 * 4$ bytes, with the extra 1 factors coming from the serialization of collection sizes. In the experiments described in this chapter the values used for the application parameters were

$C_{max} = 2$ and $V_{max} = 4$ bringing the maximum serialized size to 92 bytes. This represents a 11x bigger memory overhead as that introduced by the previously mentioned algorithms.

6.4.2.4 *Triangle Closing*

Triangle Closing is an algorithm with useful applications in social networking. In these networks vertices usually represent users and edges friendship or acquaintance relationships. By identifying edges in the graph that would close partially complete triangles and then order them based on how many neighbors share that same edge, Triangle Closing is able to identify those people that are likely part of a certain person's social circle but that person had not connected with.

Algorithm 5: Per-vertex algorithm for Triangle Closing

```
input : vertex, messages, superstep

myNeighbours = getNeighbourIds();
if superstep == 0 then
    sendMessageToNeighbours(vertex, myNeighbours);
else
    closeMap = {};
    for neighbours ∈ messages do
        for neighbour ∈ neighbours do
            if neighbour == vertex.id or neighbour ∈ myNeighbours
            then
                continue;
            closeMap[neighbour]++;
    vertex.value = getKeysOrderedByValue(closeMap);
vertex.halt();
```

Algorithm 5 details the algorithm followed by the Triangle Closing application. This application always runs for 2 supersteps. In the first superstep, each vertex sends its neighbour list to each neighbour and halts. In the second superstep, using the neighbour lists of its own neighbours, a vertex has access to 2-hop information of its neighborhood. Based on this information, a vertex V_x then proceeds to identify those vertices located 2-hops away to which it has no connection and ranks these vertices in descending order of number of connections to neighbours of V_x.

Although Triangle Closing runs for only 2 supersteps, it has a very big computational and memory overhead. Superstep 0, the lightest of the two, requires the creation of a neighbour id list and the sending of that list to every neighbour. For certain vertices this list can have several hundred, if not thousand, elements. In superstep 1, neighbour lists from all the incoming edges to a vertex are received and iterated over while updating a counter map for those vertices that appear in the 2-hop neighborhood but not the 1-hop neighborhood. This counter map is then sorted by value and an ordered list of its keys is set as the value of the vertex. Neighbour lists are composed of vertex ids which are represented as integers (4 bytes). [3] shows that active Facebook users, in 2011, had an average friend count of 190. Assuming, then, an average vertex degree of 190, each vertex would generate lists with 190 elements in superstep 0, for a serialized size of 764 bytes (adding an extra 4 bytes to keep the size of the collection) each. Seeing as a partition usually has several vertices (close to 60k in our experiments with Tuenti), the total size of generated data per partition on superstep 0 would be of about 51MB, substantially bigger than the generated data for superstep 0 in the previously described algorithms under the same conditions. For superstep 1, however, each vertex would then receive neighbour lists from each of its neighbours. Continuing with the assumption of an average vertex degree of 190, each vertex would then receive, on average, 190 lists, each with 190 vertex ids for a combined per-vertex size of close to 142KB and 36100 iterations of the innermost main loop. Totaling these figures for an entire partition, 8.1GB of serialized data would be the input of a partition with 60k vertices in superstep 1 and the vertices in that partition would perform a total of 2.16 billion iterations of the innermost main loop. Moreover, these estimates are conservative as collections in memory tend to occupy far more than their serialized size due to pointers and other management data structures.

Due to the amount of data and computation involved in this application, it is very susceptible to imbalances as even small variations in average degree of a partition can result in significant differences in terms of data or computation requirements.

6.5 EVENT PIPELINING

In this section, the effects of event pipelining in event execution time are studied by applying it to the execution of the 4 real-world applications previously described in the context of the Tuenti graph. 2 experiments were made, varying different parameters of significance to pipelining: pipeline size and event size (number of mutations per event).

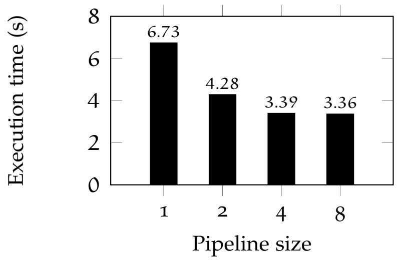
6.5.1 *Pipeline size*

The size of the pipeline determines how many events can be inside the pipeline executing concurrently (albeit at different supersteps). In this experiment, 10 events, each containing a total of 10 mutations, are submitted to a RTGiraph cluster after processing the base Tuenti graph (event 0). The time required to handle these events is measured and averaged over 5 repeated executions for each pipeline size and application. All the real-world applications described in Section 6.4.2 are considered. For pipeline sizes, the values of 1 (pipelining disabled), 2, 4 and 8 are considered. Figure 22 showcases the obtained results.

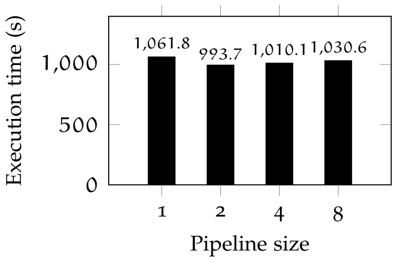
Looking at the bar charts, observed improvements to the event execution time obtained with pipelining active (pipeline size greater than 1) are between 7% (for Triangle Closing with pipeline size of 2) and 177% (for PageRank with pipeline size of 8).

SSSP and PageRank, the more lightweight applications, register the best improvements with maximums of 100% and the already mentioned 177%, respectively, both with a pipeline size of 8. Examining improvements for intermediate pipeline sizes, one notices that the benefit of increasing the pipeline size decrease exponentially. For SSSP, the increase of pipeline size from 1 to 2 provides a performance improvement of 57% while the increase from 2 to 4 and from 4 to 8 provide, respectively, an increase of 26% and 0.8%. Because PageRank execution requires slightly more time than that of SSSP, it is less affected by the base coordination and messaging overheads and is thus able to achieve greater speedups: 61% from 1 to 2, 38% from 2 to 4 and 25% from 4 to 8.

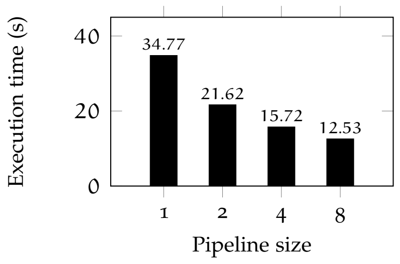
The speedups mentioned in the previous paragraph are enabled by taking advantage of the fact that with both SSSP and PageRank, the effects of mutations in the graph are quickly dampened, affecting a relatively small area around the involved vertices. This naturally causes some skewness in the execution of different events, with some events only affecting partitions owned by, for example, 2 of the 20 cluster nodes. Without pipelining, the remaining 18 nodes would be idle during this period. By allowing several events to occur simultaneously, one can exploit this skewness to considerably speedup execution. Even if an event involves the entire cluster, idle times are bound



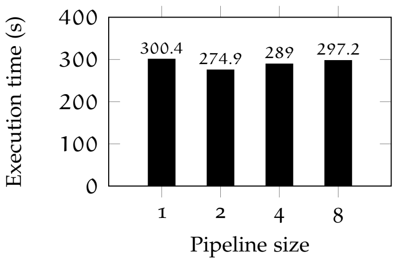
(a) Single Source Shortest Paths



(b) Triangle Closing



(c) Pagerank

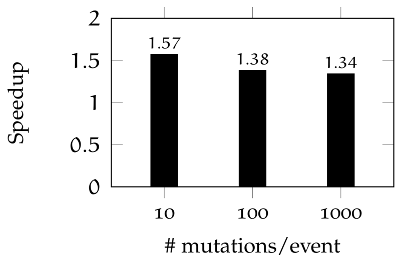


(d) Semi Clustering

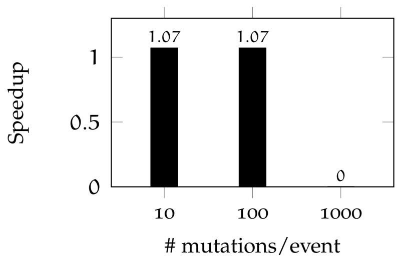
Figure 22: Event execution times as a factor of pipeline size for 4 different applications over the Tuenti graph. Measured times correspond to total execution times for 10 events with 10 mutations each. Total times do not include the time required for processing the base graph (event 0).

to occur as partitioning schemes are rarely perfect and network communication is susceptible to delays. With pipelining, that idle time in between supersteps can also be used to perform executions of a consequent event.

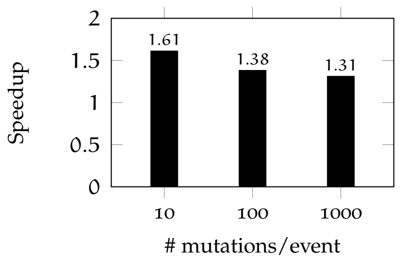
For Triangle Closing and Semi Clustering, the applications that perform the most amount of work and handle the biggest amount of data in between supersteps, subtler improvements are observed: 7% for TC and 9% for SC with a pipeline size of 2. Bigger pipeline sizes, while still achieving a relative speedup compared to no pipeline, actually suffer from some performance degradation when comparing to executions with pipeline of size 2. These observations can be explained by considering that each pipelined event requires a dedicated event executor and handles its own execution and memoization independently. Assuming for simplicity that all events have the same CPU and memory requirements, a pipeline of size 4 with 4 events executing concurrently will necessarily require 4 times the resources of a single event execution. As both Triangle Closing and Semi Clustering are at least an order of magnitude heavier than SSSP and PageRank in both CPU and memory usage, it is understandable that the system becomes more easily overloaded as the number of concurrent executions increases. This suggests that further optimizations would be



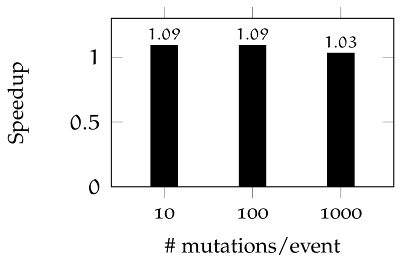
(a) Single Source Shortest Paths



(b) Triangle Closing



(c) Pagerank



(d) Semi Clustering

Figure 23: Event execution speedups as a factor of number of mutations per event for 4 different applications over the Tuenti graph. Measured speedups correspond to the quotient of total execution times with pipeline of size 1 and with pipeline of size 2 for 10 events of variable size with 10, 100 and 1000 mutations each. Total times do not include the time required for processing the base graph (event 0).

required to the pipelining mechanism to make it automatically detect the current system load and adjust the pipeline size accordingly.

6.5.2 *Event size*

The number of mutations included in an event has a direct influence on the area of impact of that event in the graph and, thus, on the amount of recomputations that have to be performed. Figure 23 highlights the speedups obtained by using an event pipeline of size 2 with 10 events containing increasing amounts of mutations: 10, 100 and 1000.

As expected, in general, the obtained speedup decreases as the size of the events increases. This is particularly evident with both SSSP and PageRank applications where the total speedup is reduced from 1.57 and 1.61 to 1.34 and 1.31, respectively. Curiously, the effect of increasing event size from 10 to 100 has a bigger impact on these applications than the increase from 100 to 1000. A possible explanation for this phenomenon lies on the different natures of exploitable idle times in non-pipelined execution:

- Non-involvement idle times — Partitions that are not involved in an event do not have anything to process and, as such, remain idle for the complete duration of the event. Between two different events, the set of involved partitions might be completely disjoint in which case the two events could theoretically be run at the same rate, with no common computational resources.

- Coordination idle times — As discussed in previous sections, Pregel requires global coordination between all the nodes in the cluster involved in the event at each superstep. Due to imperfect partitionings, variable node load, network delays, or other sources of skew, nodes might spend different amounts of time idling while waiting for this coordination to be achieved. If the node is capable of processing several events simultaneously, these otherwise wasted computational resources can exploited to accelerate the execution of other events.

As event sizes increase, the fraction of the graph affected by their mutations increases proportionally in an application specific way. Events that only affect a tiny fraction of the total partitions of a graph become, therefore, less common and, in doing so, the first category of exploitable idle times becomes increasingly rare. It might be the case that the increase from 10 to 100 mutations is sufficient, in the case of SSSP and PageRank, to shift the majority of the weight to the second category of idle times which is not able to provide the same performance improvements as the first.

In the case of the Semi Clustering and Triangle Closing algorithms, the change from 10 to 100 mutations per event appears to have a negligible impact on the event pipelining speedups. Given the bigger computational and memory requirements of these applications, the slight increase in computation time may be counteracted by longer skews and exploitable coordination idle times. From 100 to 1000, the extra load in the system does reduce the efficiency of parallel execution and the observed speedup for Semi Clustering decreases by 6%. For Triangle Closing, the change from 100 mutations to 1000 is, in fact, sufficient to exhaust the amount of memory made available to the JVM and the application crashes.

6.6 LOCAL SYNC

This section describes a series of experiments performed to explore the possible advantages of the different local sync mechanisms implemented as well as explore their limitations both under simulated and real environments.

6.6.1 *Partition-level*

The experiments described in the following sections concern the partition-level local synchronization mechanism implemented in Section 5.3.1. The first three experiments use artificially generated graphs and the skewed dummy algorithms to explore near-optimal situations for the application of partition-level local synchronization. The fourth and fifth experiments show how increasing the density of the meta-partition graph in the considered artificial graph affects the results obtained in the previous experiments. Finally, the sixth and last subsection applies this mechanism to a real-world graph and real-world applications to measure how it affects their execution.

6.6.1.1 *Disconnected partitions*

In this experiment, the Dummy Skewed application described in Section 6.4.1.1 is run on the graph shown in Figure 24, containing 4 disconnected partitions each with 500k vertices and 2M edges. Each partition is run on a different node in the cluster (thus only 4 nodes of the cluster are used in this experiment).

Firstly, a timeline for a normal execution with no local synchronization mechanism is shown in Figure 25. This timeline was constructed from event logs of actual executions. Looking at this timeline, one can observe that all partitions except partition 4 (which does the greatest amount of computation) experience significant idle periods where no useful computation is done. Final execution results are obtained for all partitions when partition 4 finishes superstep 4 and the whole event finishes.

Figure 26 shows the timeline for the same execution but with partition-level local synchronization activated. Looking at the timeline, it is immediately apparent that under these conditions, partition-level local

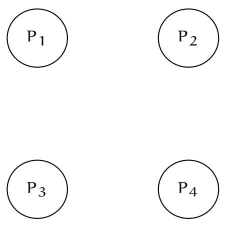


Figure 24: Graph with disconnected partitions

synchronization is able to completely remove the intermediate idle times previously witnessed and shift them to the end of the computation. While this, in itself, does not have a significant impact on the running time of the application (since in both cases the application only finishes when the slowest partition finishes), the use of local synchronization allows one to take advantage of partial final results. For instance, partition 1 is able to obtain its final results 2 times faster than partition 2 which also obtains its final results 2 times faster than partition 3 and so on. Comparing with the non-local sync execution, partition 1 is able to obtain its final results approximately 8 times faster; partition 2, 4 times faster and partition 3, 2 times faster.

Admittedly, not all applications might be able to take advantage of these partial final results as their significance might only be grasped or have true meaning when taken in context with the final results of the rest of the graph. However, some use cases can indeed benefit from having partial final results made available as soon as possible. Some examples are:

- Visualization tools might start prerendering this partial data to reduce the amount of work that would otherwise have to be done if rendering only started when the entire result dataset was available.
- Applications that analyze graphs looking for special types of entities that require the triggering of special actions can use these partial final results to trigger those actions with confidence as soon as possible instead of waiting until the whole dataset is analyzed. A good example of a set of such applications are those that are used to identify malicious behavior: fake users in social networks, credit card transaction fraud, terrorist activity. Being able to take prompt action in these cases is of the utmost importance.

The benefit of local synchronization in these kinds of graphs is accentuated when used in conjunction with the pipelining mechanism previously analyzed. Considering that a new event arrives to the system adding 10k edges to vertices in partition 1, Figure 27

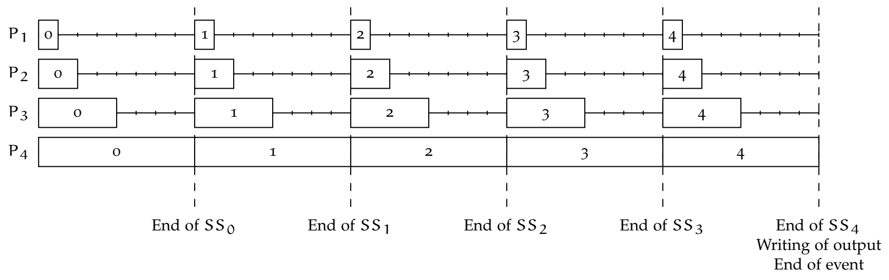


Figure 25: Normal execution timeline with disconnected partitions.

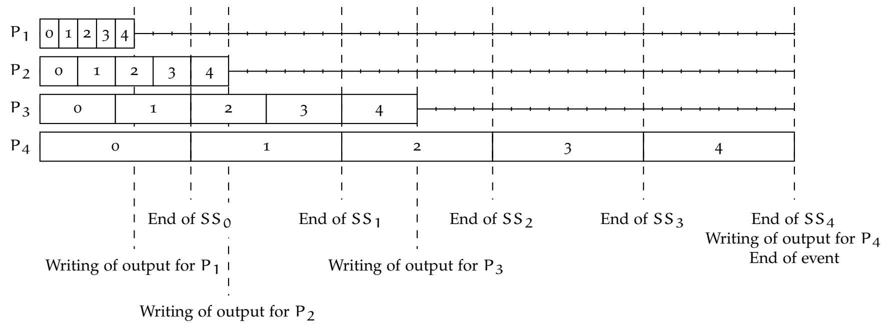


Figure 26: Partition-level local sync execution timeline with disconnected partitions.

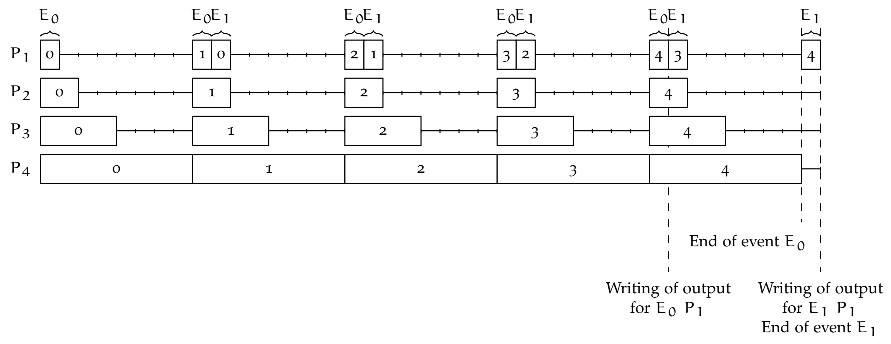


Figure 27: Normal execution timeline with disconnected partitions. A new event (E_1) is considered under a pipelined execution. E_1 only involves partition 1.

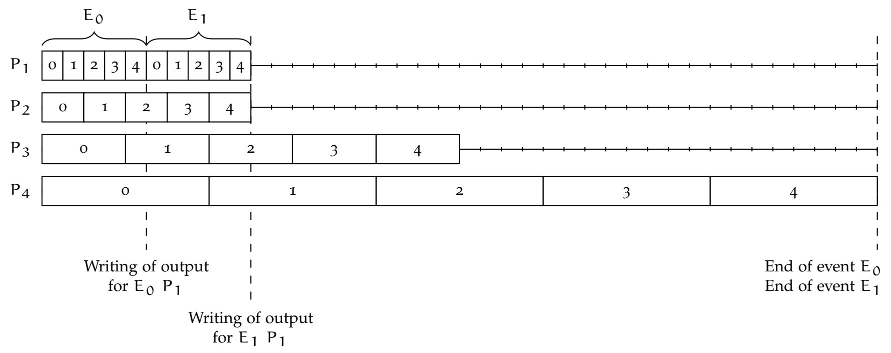


Figure 28: Partition-level localsync execution timeline with disconnected partitions. A new event (E_1) is considered under a pipelined execution. E_1 only involves partition 1.

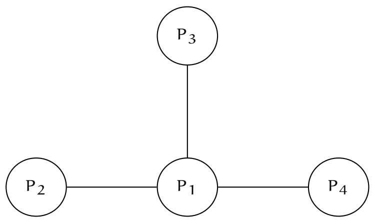


Figure 29: Graph with sparsely connected partitions

shows the timeline for an execution with pipelining activated. Similarly, Figure 28 shows the timeline for the same execution but now with partition-level local synchronization activated. Final results for event 1 are made available at partition 1 even before the slowest ones are able to produce their final results for event 0. Therefore, we can see that speedups gained by certain partitions for previous events are directly transferred to the execution of consequent events in the same partitions.

While the analysis done in this section shows that partition-level local sync can obtain significant speedups in the presence of disconnected components in the graph, one might argue that the same benefits could have been obtained by identifying a priori the disconnected nature of the graph and submitting the different components as multiple applications. In fact, this would indeed produce similar (if not identical) execution timelines. However, accomplishing that would require having previous knowledge about the graph structure which is not always possible. By working in tandem, the partitioning mechanism of RTGiraph along with partition-level local synchronization are able to automate this process.

Furthermore, while this example examines the simple case of having 4 independent partitions, the obtained results can be extended to every graph having 4 connected components even if these components consist of multiple interconnected partitions.

6.6.1.2 *Sparsely connected partitions and consistent skew*

Despite the good results obtained in Section 6.6.1.1, it is not common for graphs to have disconnected partition meta-graphs. In this experiment, we extend the previous one by including extra inter-partition connections in the artificial graph. In particular, 1k edges are added between partition 1 and each of the partitions 2, 3 and 4. The resulting graph is shown in Figure 29.

Since non-localsync execution completely ignores dependencies between partitions, the execution timeline for this graph would be exactly the same as in Figure 25.

With partition-level localsync, however, the added dependencies between partitions will now restrict the amount of computation that can be done ahead of the stragglers as shown in the timeline in Fig-

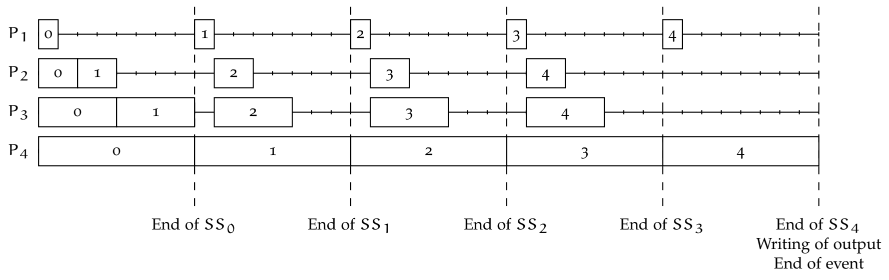


Figure 30: Partition-level localsync execution timeline with sparsely connected partitions and consistent skew.

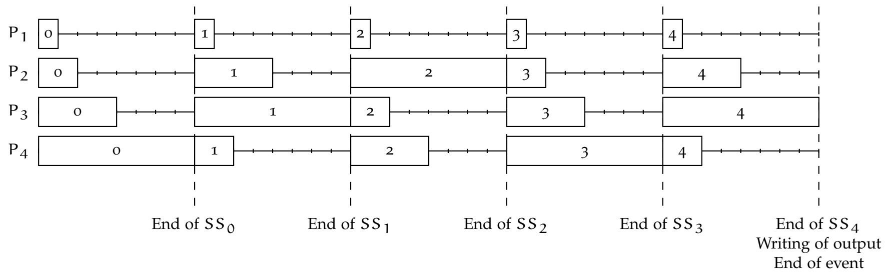


Figure 31: Normal execution timeline with sparsely connected partitions and variable skew.

ure 30. While partition 2 and 3 are able to compress the initial idle period between the computations of superstep 0 and superstep 1, that compression only enables them to stay slightly ahead of the straggler computation of partition 4 and new idle periods soon appear after superstep 1. Partition 1, the one doing the least amount of computation, is unable to do any kind of computation ahead of partition 4 as partition 4 is now a direct dependency. Furthermore, any chance that partitions 2 and 3 had of outputting their partial final results ahead of the others when finishing computation of superstep 4 is now eliminated because these partitions cannot independently guarantee (without application-specific knowledge) that they won't receive any further messages from partition 1 when it starts computation of superstep 4. Such an event would then require the former partitions to do extra computation in superstep 5.

6.6.1.3 *Sparsely connected partitions and variable skew*

In the previous experiment, partition-level local sync was shown to be unable to do any significant improvement in the execution of sparsely connected meta-partition graphs with a constant skew. In this experiment, partition-level local sync will be tested against the same graph previously used but now using the Dummy Skewed Alternating application described in Section 6.4.1.2. With this application, the partition with greatest computational load will vary in each superstep.

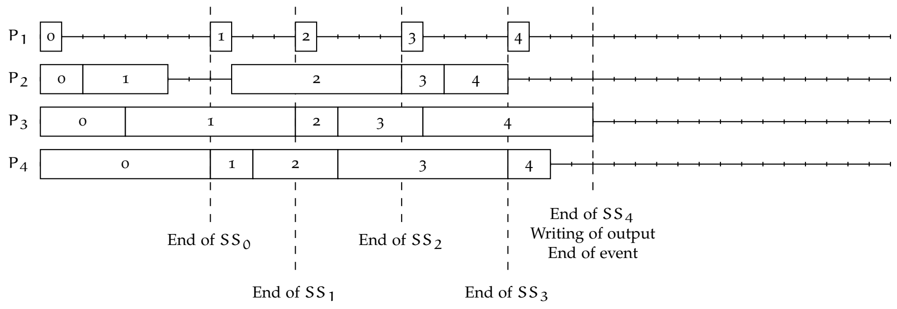


Figure 32: Partition-level localsync execution timeline with sparsely connected partitions and variable skew.

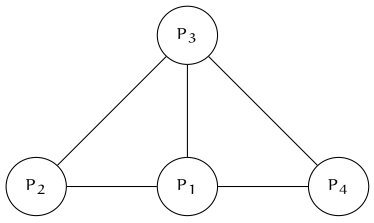


Figure 33: Graph with densely connected partitions

As before, Figure 31 and Figure 32 show the timelines for, respectively, non-local sync and partition-level local sync executions. Under the presence of variable skew, partition-level local synchronization is able to reduce the total running time of the application by fitting part of the execution of the stragglers in the otherwise idle period of the previous superstep. With the particular conditions here described, the average registered speedup was of approximately 1.375 (200 seconds of non-localsync vs 145.5 seconds of partition-level localsync). Considering an application following these conditions and that runs, in average, for 60 minutes, the registered speedup would bring the total execution time of that application down to 43 minutes.

6.6.1.4 *Densely connected partitions and variable skew*

Having seen that partition-level localsync is able to reduce the total computation time for graphs with sparsely connected partitions under a variable skew per node, this experiment explores what occurs as the partition meta-graph becomes increasingly connected. To that end, 1k edges are added from partition 3 to each of partition 1 and 4 resulting in the final graph shown in Figure 33.

Once again, the change in the partition dependency sets has no effect in normal non-localsync execution so the timeline of the previous experiment shown in Figure 31 also applies to this experiment. For partition-level localsync, the execution timeline is shown in Figure 34.

Looking at the timeline, one can witness a compression of the runtime of superstep 3 as the more skewed computation of partition 4 takes advantage of the otherwise idle time of superstep 2. However, in

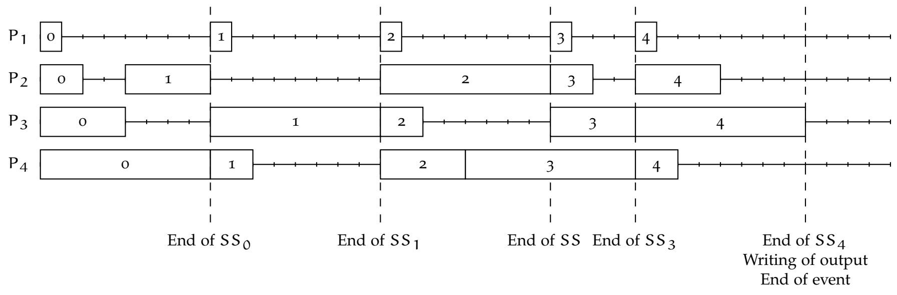


Figure 34: Partition-level localsync execution timeline with densely connected partitions and variable skew.

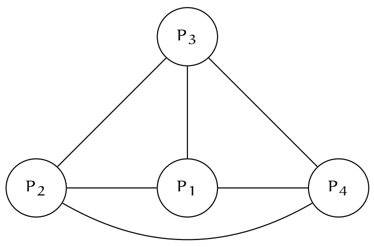


Figure 35: Graph with completely connected partitions

this case, that is the only time a compression of runtime is achieved. In the previous experiment, such a compression was achieved for each of the supersteps following superstep 0. The execution time for the non-localsync run remains the same as in the last experiment: approximately 200 seconds. However, partition-level localsync now takes approximately 190 seconds against the 140 seconds that it had accomplished with a sparser partition meta-graph. Therefore, with a more densely connected graph, the speedup of partition-level localsync is greatly impacted, being reduced to a mere 1.053.

6.6.1.5 *Completely connected partitions and variable skew*

The final experiment of partition-level local synchronization with artificial graphs will explore its effects when in the presence of a completely connected partition meta-graph. To that end, the graph considered in the previous experiment is extended by adding another 1k edges from partition 2 to partition 4. The final graph is shown in Figure 35.

The timeline for partition-level localsync is shown in Figure 36. Examining the two figures, it becomes obvious that partition-level local sync is unable to provide any significant advantages over normal execution for completely connected partition meta-graphs even in the presence of variable skew. As both executions share the same execution timeline, it comes as no surprise that their runtimes are also equal to one another, at the approximate 200 seconds witnessed in previous experiments.

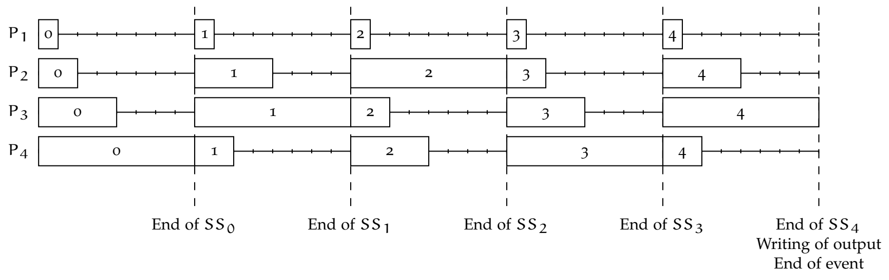


Figure 36: Partition-level localsync execution timeline with completely connected partitions and variable skew.

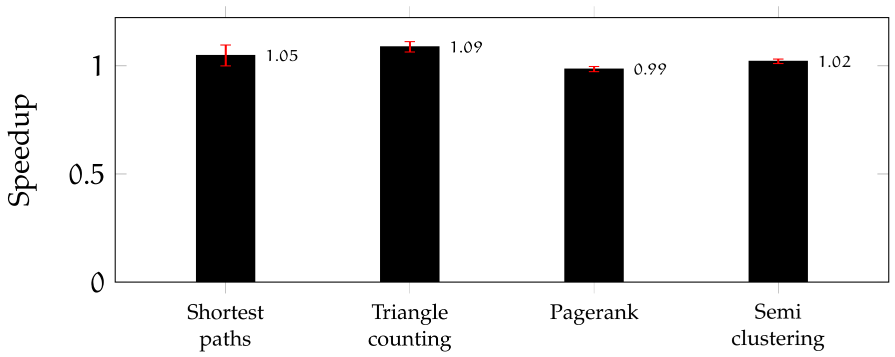


Figure 37: Average speedup obtained with executions of the algorithms identified in the x-axis in the Tuenti graph with partition-level localsync. Non-localsync mechanism used as reference base for speedup computation. Error bars represent standard deviation.

6.6.1.6 *Tuenti graph*

Figure 37 shows the speedups obtained by using the partition-level local synchronization mechanism with real applications on the real-world Tuenti graph. This graph, as most social graphs, follows a small-world structure [33] in which most nodes are connected to others through short paths. In this scenario, partition-level local synchronization does not seem to provide any significant improvements over the traditional non-localsync, with the average speedup remaining at 1 or so near to it as not to make a discernible difference. Examining the structure of the graph, the reason for this becomes obvious: the partition meta-graph is completely connected, mimicking the conditions described in Section 6.6.1.5.

The Triangle Closing application was the one where the biggest speedups were registered. These speedups were consistent across the entire population of measurement samples and consisted in runtime differences of up to 50 seconds, suggesting that something else besides usual performance fluctuation was at play. Examination of the execution logs and profiling with YourKit revealed that, while executing the Triangle Closing application, the cluster operates very close to maximum capacity in terms of memory. This is justifiable by the large size that messages under this application can take as described in Section 6.4.2.4. Under these conditions, the JVM's Garbage Collector performs frequent collection pauses, occasionally triggering ZooKeeper channel disconnections whose reconnection may take up to 10 seconds. The non-localsync mechanism relies on ZooKeeper coordination to break the start barrier of the next superstep and only when this occurs launches local partitions for execution of that superstep. With the partition-level localsync mechanism, a prefetching effect is achieved where the launching of a partition for execution of the next superstep occurs as soon as a message is received from the last dependency of that partition. Because this logic completely bypasses ZooKeeper coordination, partition-level localsync is able to take advantage of this time, otherwise completely dedicated to inter-node coordination, for next superstep loading and computation.

Identical analysis for the remaining applications shows that this prefetching effect is also present in their execution. However, because the total execution times are smaller and/or the inter-node coordination period is less prone to excessive delays, its effect is not so pronounced as in the Triangle Closing case.

Another important result from this experiment is the fact that this partition-level local synchronization mechanism does not appear to negatively impact the execution of applications when the graph is not favorable. As such, and given the results obtained with the previous experiments, it could be used in place of the traditional one, improving performance in those cases where the partition meta-graph is sparser, without penalizing less favorable executions.

6.6.2 *Vertex-level*

The experiments described in the following sections concern the vertex-level local synchronization mechanism implemented in Section 5.3.2. They reenact some of the scenarios considered in the evaluation of partition-level local synchronization to investigate to what extent vertex-level local synchronization is able to improve upon the results obtained with the previous local synchronization scheme.

6.6.2.1 *Densely connected partitions*

Having shown in Section 6.6.1.4 that partition-level local synchronization loses effectiveness when the partition meta-graph starts getting heavily connected, in this experiment, the effectiveness of vertex-level local synchronization will be tested under the same conditions to see if any further benefits can be obtained.

The average execution time with vertex-level local sync was of approximately 174 seconds, a 9% improvement over the execution time obtained with partition-level local sync (approximately 190 seconds) and a 16% improvement over that obtained without any local synchronization (approximately 201 seconds). The reason for this improvement can be seen in Figure 38: part of the computation to be done in supersteps following superstep 0 is done in incomplete passes using the idle time of the previous superstep.

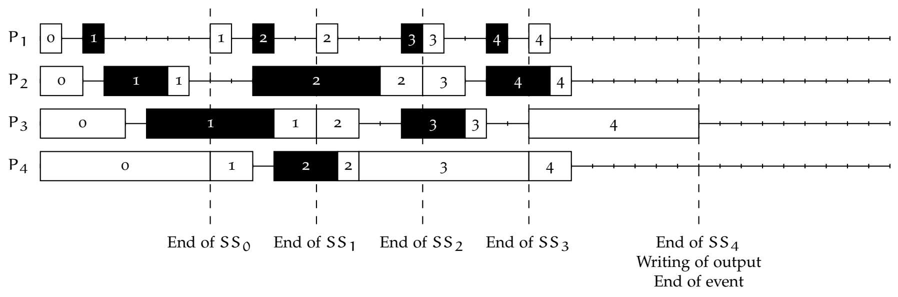


Figure 38: Vertex-level localsync execution timeline with densely connected partitions and variable skew. Dark-colored bars represent preemptive incomplete passes while light-colored bars represent final passes.

6.6.2.2 *Completely connected partitions*

Given the improvement registered in the previous experiment compared to the two other execution mechanisms, the experiment described in this section attempts to determine if vertex-level local synchronization is also able to provide performance benefits when the partition meta-graph is fully connected as happened in the experiment described in Section 6.6.1.5.

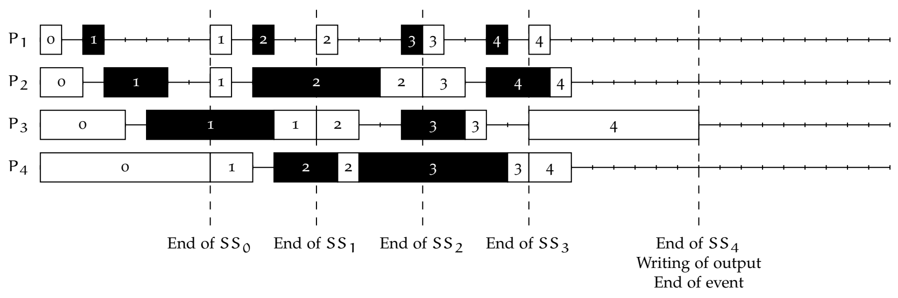


Figure 39: Vertex-level localsync execution timeline with completely connected partitions and variable skew. Dark-colored bars represent preemptive incomplete passes while light-colored bars represent final passes.

Indeed, the average execution time with vertex-level local sync under the conditions of the aforementioned experiment averaged at 174 seconds, the same value registered for the previous experiment with densely connected partitions. Since, under these conditions, there was no discernible difference between the execution times of both non-localsync and partition-level localsync, the result here obtained constitutes a 16% improvement over those execution mechanisms. Once again, this improvement can be explained by the use of idle time in the previous superstep to perform part of the computation for the next as seen in Figure 39.

6.6.2.3 *Tuenti graph*

Similarly to the experiment described in Section 6.6.1.6 for partition-level localsync, in this section we examine the speedups obtained with vertex-level localsync during standard executions of real applications in the Tuenti graph. These speedups are illustrated in Figure 40.

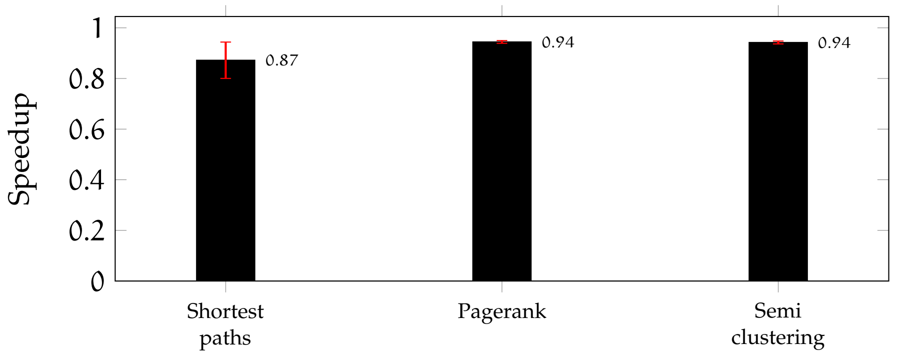


Figure 40: Average speedup obtained with executions of the algorithms identified in the x-axis in the Tuenti graph with vertex-level localsync. Non-localsync mechanism is used as reference base for speedup computation. Error bars represent standard deviation.

The obtained results suggest that vertex-level local synchronization has a non-negligible negative impact in the execution time of applications in the Tuenti graph. This can be explained by 2 factors:

- Data merging — Due to the 2-pass system, vertex-level local synchronization has to merge state and deltas generated by the two different passes. Since this data is stored in arrays of bytes, the merge requires the allocation of a new byte array to hold the final result for both state and deltas. This essentially doubles the amount of memory necessary at the time of the merge. Furthermore, when handling a great amount of vertices and messages, the allocation and freeing of these byte arrays can have a significant impact in the performance of the Garbage Collector. In addition, since data needs to be stored in these arrays ordered by vertex id, the merging of the intermediate arrays cannot be made with simple copy mechanisms but has to rely on a 2-way merge algorithm with a time complexity of $O(N + M)$ with N being the size of the array of the first pass and M of the array of the second pass.

- Vertex skipping — Because, when doing the first pass, some of the partition's dependencies have not yet been satisfied, not all of its vertices can compute. The checking of which vertices can compute or not involves checking the edges of that vertex and determining if the target partition is in the set of finished partitions or not. If all edges of the vertex satisfy this condition, the vertex can proceed with the computation. Otherwise, its computation has to be delayed to the second pass. This checking has a time complexity of $O(E)$ where E is the number of edges known to the partition.

In algorithms where only a fraction of the vertices compute in each superstep and where the per-vertex computation is inexpensive, the weight of this overhead is quite significant, specially if a pass only results in the computation of a reduced number of vertices. This is the case of the Shortest Paths application where we see an average decrease in performance of 13%. In applications where per-vertex computation is heavier or the majority of the vertices are involved in the computation at each superstep, this overhead becomes less significant as happens with Pagerank or Semi Clustering.

As detailed in Section 6.6.1.6, the triangle counting application was already running close to the maximum memory limit of the cluster due to the great size of the messages it sends per vertex. Due to the extra memory necessary to perform the merge of the data from the two passes, this memory limit is reached and the application aborts. Therefore, its speedup is not represented in Figure 40.

6.6.2.4 *Tuenti graph with artificial alternating skew*

The results of the previous experiment suggest that a normal execution of the applications in the cluster does not contain enough alternating skew and, thus, does not provide a favorable environment for obtaining performance improvements with the vertex-level local synchronization mechanism. In this section, the previous experiment is repeated but now introducing extra computational load at each per-vertex computation using the same mechanism as that detailed in Section 6.4.1.2. This extra load is integrated with the normal computation of the application , thus allowing one to keep the original execution and communication pattern while manipulating the idle times between supersteps.

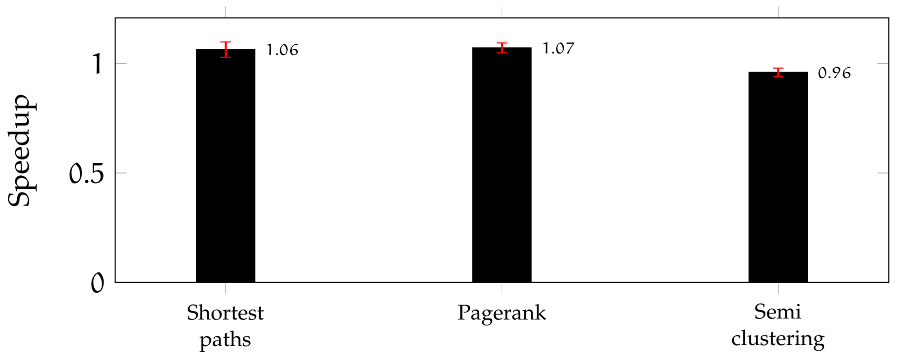


Figure 41: Average speedup obtained with executions of the algorithms identified in the x-axis in the Tuenti graph with vertex-level localsync. Artificial alternating skews are added to the per-vertex computation of each application. Non-localsync mechanism is used as reference base for speedup computation. Error bars represent standard deviation.

Figure 41 showcases the new speedups obtained under this environment with alternating skews. Looking at the figure, it is evident that, in this scenario, vertex-level local synchronization is able to counteract the observed overhead in the previous experiments for SSSP and PageRank, obtaining actual improvements of, respectively, 6% and 9% of the total execution time. Semi Clustering still suffers from the high overhead of this synchronization mechanism but sees this overhead reduced from 6% to 4% of total execution time.

7 CONCLUSIONS

With the exponential growth of graph-structured data, the need for efficient large scale graph processing mechanisms became increasingly dire. Systems such as Giraph, Hama and Graphlab rose up to the challenge of efficiently handling batch computations over graphs by using APIs and storage and traversal mechanisms specifically tailored to these datasets. However, graph data is naturally evolving, constantly receiving vertex or edge mutations and, as a result, analytics over the graphs have to be constantly kept up to date. The aforementioned systems waste precious time and computational resources by recomputing the entirety of the graph when such a mutation arrives. In a competitive environment where every second counts, being able to obtain results with lower latency is of vital importance. With that in mind, RTGiraph was designed to combine the large scale graph processing power of its predecessor (Giraph) with concepts of incremental computation. In doing so, it is able to reuse the results of previous computations to greatly speedup the updating of analytics as the underlying graph mutates.

Despite the performance improvements obtained with RTGiraph, its reliance on global synchronization for computation advancement inherited from Giraph and the Pregel model make it particularly vulnerable to performance degradation in the presence of computational skew or network jitter. In these situations, one or more nodes lag behind and delay the processing speed of the entire cluster, with the processing power of the remaining nodes being wasted through idling.

In this thesis, several mechanisms for taking advantage of these idle times were considered, designed, implemented and evaluated in the context of the RTGiraph system.

The first, event pipelining, exploits the streaming nature of graph mutations to concurrently process several consecutive events, albeit at different computation stages (supersteps). Otherwise idle nodes can now dedicate their computational resources to the ahead-of-time processing of events following the one in which they are currently blocked waiting for global coordination. The implementation of this mechanism required significant changes to the global synchronization, event execution and memoization modules of RTGiraph. Evaluations showed as much as 177% performance improvements when using event pipelining with more lightweight applications such as Single Source Shortest Paths or PageRank. However, heavier applications such as Triangle Closing and Semi Clustering registered more conservative improvements of around 7% and 9%, respectively. This difference in speedups can be explained by the fact that the resources needed scale in a direct proportion with the size of the pipeline and, thus, with heavier computations, nodes get more easily overloaded decreasing the probability of having idle times to exploit or the effectiveness with which these are exploited.

The second mechanism, partition-level local synchronization, attempts to tackle the problem not by doing extra useful computation during idle times but by removing or considerably reducing them. To achieve this, the global synchronization barriers native to Pregel-based systems are replaced with local synchronization barriers. These local barriers, with the assistance of a dependency tracking module, block progress of a partition execution until such a time as all the dependencies of that partition have finished the previous superstep. Thus partitions (and, by extension, nodes) now only have to wait for a subset of all partitions to finish before being able to continue execution themselves. Implementing this mechanism required a complete overhaul of the event execution module, splitting it into two separate execution layers. A dependency tracking mechanism was also designed that infers dependencies from the graph structure and keeps track of which partitions have finished executing a certain superstep at any one time. Evaluation with artificial graphs and algorithms has shown that the performance benefits of these local synchronization barriers are greatly dependent on the connectivity density of the partition meta-graph and on the type and amount of skew observed in the nodes of the cluster. With disconnected components in the partition meta-graph, local synchronization is able to provide partial final results considerably faster than a normal execution with global synchronization (a speedup of up to 8x for partial results was observed in experiments). As the meta-graph becomes increasingly connected, the obtained benefit of the mechanism decreases with experiments showing a decrease from 37.5% performance improvement, to 5.3% and, with a fully connected meta-graph, to approximately 0%.

The third mechanism, vertex-level local synchronization, builds upon the previous local synchronization mechanism by allowing the starting of computation of a partition even before all of its dependencies have finished. This is done by allowing the execution of up to 2 passes over a partition in every superstep. If, when trying to schedule execution of a partition for the next superstep, the system notices that a configurable amount of that partition's dependencies (but not all) have already finished, it can allow the partial computation of those vertices in the partition that are sure to already have all the required information for computing the next superstep correctly. When the remainder of the dependencies finish, a second final pass is scheduled to compute all the previously skipped vertices. For this mechanism to work, the 2-pass logic had to be integrated into the event and partition execution modules, requiring vertex dependency analysis and merging of partial computation data. Vertex-level local synchronization was shown to be able to obtain further improvements than those obtained by partition-level local synchronization alone. In particular, in the same conditions under which the latter mechanism failed to provide any improvement, the former obtained a 16% improvement in execution time. Using it with a large real-world graph and applications, however, has shown that the merging and vertex-level dependency tracking introduces significant overhead under normal non-skewed conditions. In the presence of variable skew, however, this overhead is compensated and overall improvements of up to 7% were achieved.

In summary, all implemented mechanisms have shown ability to improve the latency of graph processing applications in particular scenarios. Event pipelining has shown significant improvements in the processing of multiple events simultaneously while partition-level local synchronization has shown to be very effective in situations with variable skew and with graphs whose partition meta-graph contains disconnected components or is very sparse. In addition, it does not introduce significant overhead to non-favorable executions. Finally, vertex-level local synchronization can speedup executions in graphs with densely or completely connected partition meta-graphs in the presence of alternating skew. However, the 2-pass mechanism introduces non-negligible overhead which negatively affects performance in scenarios where it is not very effective (no or very little skew).

7.1 FUTURE WORK

Event Pipelining

While event pipelining has shown significant speedups in the described experiments, there is still room for improvement. In particular, experiments have shown that the obtained speedups vary greatly according to the characteristics of the application being run over the graph and that these variations are most likely the result of overloading of the computational resources of the system. Ideally, the pipelining module should be able to, based on information from previous events and/or executions of the same algorithm, dynamically assess the benefit of different pipeline sizes and automatically choose the one that offers the best benefit while keeping concurrent computation size inside the capabilities of the cluster.

Partition-level local synchronization

When it comes to partition-level local synchronization, the biggest possibilities for improvement come from the reduction of messages sent. Although no significant overhead was registered in the experiments performed in this thesis, partition-level local synchronization does require a larger amount of messages to be sent to be able to keep track of when the dependencies of a partition finish a superstep (even partitions that did not perform any work in a superstep need to send empty messages to other partitions they are connected to).

Further work could focus on strategies to reduce this impact, perhaps by extending the application API and making use of application-specific knowledge to predict the actual number of real messages to be received in a particular superstep similar to what was done in some of the research described in Section 2.6.

Aggregation of individual messages in batches could also be analyzed to try and determine if communication savings could be achieved without too much sacrifice to the latency of the dependency tracking.

Finally, it is also not clear if other partitioning strategies other than the one employed by Spinner (the partitioning mechanism used for partitioning the Tuenti graph) could result in more favorable partition meta-graphs that are not as connected.

Vertex-level local synchronization

For vertex-level synchronization, the most obvious area of improvement concerns the reduction of overhead with the 2-pass mechanism. The use of memory and union efficient indices (such as bloom filters) of vertex dependencies might be able to tackle part of the overhead associated with checking which vertices can run in an incomplete pass. In addition, research into ways of doing the 2-way merge of state and deltas in-place, without requiring the creation of intermediate temporary byte arrays, would reduce the amount of allocations done and eventual pause times of the JVM garbage collection system.

A smarter scheduling of first pass executions might also mitigate the problem of having one of the two passes perform very little amount of work and, thus, introducing extra overhead for very little gain.

BIBLIOGRAPHY

[1] Vicki H. Allan et al. "Software pipelining". In: *ACM Computing Surveys* 27.3 (Sept. 1995), pp. 367–432. ISSN: 03600300. DOI: 10.1145/212094.212131. URL: http://dl.acm.org/citation.cfm?id=212094.212131 (cit. on p. 12).

[2] Amazon Web Services. *Amazon EC2 Instances*. URL: http://aws.amazon.com/ec2/instance-types/ (visited on 06/06/2014) (cit. on p. 53).

[3] Lars Backstrom. *Anatomy of Facebook*. 2011. URL: https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859 (visited on 06/06/2014) (cit. on p. 62).

[4] D P Bertsekas and J N Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Vol. 23. Prentice Hall, 1989, p. 715. ISBN: 0136487009. URL: http://portal.acm.org/citation.cfm?id=59912 (cit. on p. 12).

[5] Pramod Bhatotia et al. "Incoop: MapReduce for incremental computations". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11* (2011), pp. 1–14. ISSN: 1450309763. DOI: 10.1145/2038916.2038923. URL: http://dl.acm.org/citation.cfm?doid=2038916.2038923 (cit. on p. 11).

[6] Business Wire. *Record-Setting Holiday Season for Amazon Prime*. 2013. URL: http://www.businesswire.com/news/home/20131226005066/en/Record-Setting-Holiday-Season-Amazon-Prime (visited on 06/06/2014) (cit. on p. 3).

[7] Zhuhua Cai, Dionysios Logothetis and Georgos Siganos. "Facilitating real-time graph mining". In: *Proceedings of the fourth international workshop on Cloud data management - CloudDB '12*. New York, New York, USA: ACM Press, Oct. 2012, p. 1. ISBN: 9781450317085. DOI: 10.1145/2390021.2390023. URL: http://dl.acm.org/citation.cfm?id=2390021.2390023 (cit. on p. 11).

[8] Raymond Cheng et al. "Kineograph : Taking the Pulse of a Fast-Changing and Connected World". In: *Proceedings of the 7th ACM european conference on Computer Systems EuroSys 12* (2012), pp. 85–98. DOI: 10.1145/2168836.2168846. URL: http://www.raymondcheng.net/download/euro065-cheng.pdf (cit. on p. 11).

[9] Avery Ching. *Scaling Apache Giraph to a trillion edges*. 2013. URL: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920 (visited on 06/06/2014) (cit. on pp. 3, 10).

[10] Amr Fahmy and Abdelsalam Heddaya. “Management of Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk”. In: (July 1996). URL: http://dl.acm.org/citation.cfm?id=864302 (cit. on p. 14).

[11] J A Fisher and R Rau. “Instruction-level parallel processing.” In: *Science (New York, N.Y.)* 253.5025 (1991), pp. 1233–1241. ISSN: 0036-8075. DOI: 10.1126/science.253.5025.1233 (cit. on p. 12).

[12] JE Gonzalez, Y Low and H Gu. “Powergraph: Distributed graph-parallel computation on natural graphs”. In: *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), pp. 17–30. URL: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf (cit. on p. 39).

[13] Jakob Homan. *Apache Giraph, a framework for large-scale graph processing on Hadoop, reaches 0.1 milestone*. 2012. URL: https://engineering.linkedin.com/open-source/apache-giraph-framework-large-scale-graph-processing-hadoop-reaches-01-milestone (visited on 06/06/2014) (cit. on p. 10).

[14] Chu Shik Jhon Jin-soo Kim Soonhoi Ha. “Relaxed Barrier Synchronization for the BSP Model of Computation on Message-passing Architectures”. In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.8149 (cit. on p. 14).

[15] M. Lam. “Software pipelining: an effective scheduling technique for VLIW machines”. In: *ACM SIGPLAN Notices* 23.7 (July 1988), pp. 318–328. ISSN: 03621340. DOI: 10.1145/960116.54022. URL: http://dl.acm.org/citation.cfm?id=960116.54022 (cit. on p. 12).

[16] Yucheng Low et al. “Distributed GraphLab: a framework for machine learning and data mining in the cloud”. In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727. ISSN: 2150-8097. arXiv: 1204.6078. URL: http://dl.acm.org/citation.cfm?id=2212354 (cit. on pp. 3, 39).

[17] Grzegorz Malewicz et al. *Pregel*. SIGMOD '10. ACM Press, 2010, p. 135. ISBN: 9781450300322. DOI: 10.1145/1807167.1807184. URL: http://portal.acm.org/citation.cfm?doid=1807167.1807184 (cit. on pp. 3, 8, 59).

[18] Claudio Martella, Dionysios Logothetis and Georgos Siganos. “Spinner: Scalable Graph Partitioning for the Cloud”. In: *arXiv preprint arXiv:1404.3861* (2014). arXiv: 1404.3861. URL: http://arxiv.org/abs/1404.3861 (cit. on p. 55).

[19] Nathan Marz. *A Storm is coming: more details and plans for release*. 2011. URL: https://blog.twitter.com/2011/storm-coming-more-details-and-plans-release (visited on 09/06/2014) (cit. on p. 12).

[20] Karen Miller. *Instruction Level Parallelism: Pipelining*. 2006. URL: http://pages.cs.wisc.edu/~cs354-2/beyond354/pipelining.html (visited on 09/06/2014) (cit. on p. 12).

[21] Janak H. Patel and Edward S. Davidson. "Improving the throughput of a pipeline by insertion of delays". In: *ACM SIGARCH Computer Architecture News* 4.4 (Jan. 1976), pp. 159–164. ISSN: 01635964. DOI: 10.1145/633617.803575. URL: http://dl.acm.org/citation.cfm?id=633617.803575 (cit. on p. 12).

[22] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *DBMS*. OSDI'10 2006 (2010), pp. 1–15. URL: http://www.usenix.org/events/osdi10/tech/full%5C_papers/Peng.pdf (cit. on p. 10).

[23] G. Ramalingam and Thomas Reps. "A categorized bibliography on incremental computation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*. New York, New York, USA: ACM Press, Mar. 1993, pp. 502–510. ISBN: 0897915607. DOI: 10.1145/158511.158710. URL: http://dl.acm.org/citation.cfm?id=158511.158710 (cit. on p. 10).

[24] James F. Philbin Richard D. Alpert Richard D. "cBSP: Zero-Cost Synchronization in a Modified BSP Model". In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.7784 (cit. on p. 14).

[25] Semih Salihoglu and Jennifer Widom. "GPS". In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management - SSDBM*. New York, New York, USA: ACM Press, July 2013, p. 1. ISBN: 9781450319218. DOI: 10.1145/2484838.2484843. URL: http://dl.acm.org/citation.cfm?id=2484838.2484843 (cit. on p. 9).

[26] Telefónica. *Grafos.ml*. URL: http://grafos.ml/ (visited on 06/06/2014) (cit. on p. 11).

[27] The Apache Software Foundation. *Apache Giraph*. URL: http://giraph.apache.org/ (visited on 06/06/2014) (cit. on p. 9).

[28] The Apache Software Foundation. *Hama - a general BSP framework on top of Hadoop*. URL: https://hama.apache.org/ (visited on 10/06/2014) (cit. on p. 9).

[29] The Apache Software Foundation. *S4: Distributed Stream Computing Platform*. URL: http://incubator.apache.org/s4/ (visited on 09/06/2014) (cit. on p. 12).

[30] The Apache Software Foundation. *Storm, distributed and fault-tolerant realtime computation*. URL: http://storm.incubator.apache.org/ (visited on 09/06/2014) (cit. on p. 12).

[31] Leslie G. Valiant. *A bridging model for parallel computation*. 1990. DOI: 10.1145/79173.79181 (cit. on pp. 3, 8).

[32] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN”. In: *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*. New York, New York, USA: ACM Press, Oct. 2013, pp. 1–16. ISBN: 9781450324281. DOI: 10.1145/2523616.2523633. URL: http://dl.acm.org/citation.cfm?id=2523616.2523633 (cit. on p. 54).

[33] D J Watts and S H Strogatz. “Collective dynamics of 'small-world' networks.” In: *Nature* 393.6684 (1998), pp. 440–442. ISSN: 0028-0836. DOI: 10.1038/30918. arXiv: 0803.0939v1 (cit. on p. 74).

[34] Tao Zou et al. “Making time-stepped applications tick in the cloud”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*. New York, New York, USA: ACM Press, Oct. 2011, pp. 1–14. ISBN: 9781450309769. DOI: 10.1145/2038916.2038936. URL: http://dl.acm.org/citation.cfm?id=2038916.2038936 (cit. on p. 15).