University of Zurich[UZH]

Graph Partitioning for Signal/Collect

Master Thesis July 15, 2013

Thomas Keller
of Müselbach SG, Switzerland

Student-ID: 07-707-383
thomas.keller2@uzh.ch

Advisor:
Coralia-Mihaela Verman

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First of all, I wish to thank my advisor Coralia-Mihaela Verman for the help during my work on this thesis. I am also sincerely grateful to Philip Stutz, the main author of Signal/Collect, for his valuable advice. I would like to thank Professor Abraham Bernstein PhD for giving me the possibility to write my thesis at his Dynamic and Distributed Information Systems Group at the University of Zurich. Cosmin Basca, Lorenz Fischer, Hanspeter Kunz, and Thomas Scharrenbach resolved problems with the evaluation environment. Finally, this thesis would not have been possible without the support from my family.

Zusammenfassung

Signal/Collect ist ein Programmiermodell und System zur Ausführung von Algorithmen auf Graphen. Wird ein Graph auf mehrere Rechner verteilt, erhöht die Kommunikation zwischen den Knoten des Graphen die Laufzeit des Systems. In dieser Arbeit wird Signal/Collect erweitert, um Algorithmen zu ermöglichen, welche die Knoten während der Laufzeit zwischen den ausführenden Rechnern so verschieben, dass die Zahl der Nachrichten zwischen den Rechnern reduziert wird. Von mehreren Algorithmen wird der beste eingehend diskutiert. Weiterführende Evaluationen zeigen, dass jener Algorithmus die Laufzeit in einem von zwei Fällen reduziert. Allerdings sind die durchgeführten Untersuchungen nicht ausreichend, um ein abschliessendes Urteil zu fällen.

Abstract

Signal/Collect is a vertex-centric programming model and framework for graph processing. The communication between the vertices of a graph impairs the performance of distributed framework executions due to message serializations and network limitations. In this thesis, Signal/Collect is extended to support algorithms that reduce the number of remote messages by moving vertices between compute nodes during the computation. Several algorithms are evaluated and the best performing candidate is discussed in detail. The evaluation results indicate an improvement of the runtime performance in one of two cases. However, the performed evaluations are not sufficient to draw final conclusions about the implemented approach.

Table of Contents

1

Introduction

Signal/Collect [Stutz et al., 2010][Stutz et al., 2013] is a programming model and framework for scalable synchronous and asynchronous graph processing. Graph processing means that an algorithm is executed on a graph. The vertex-centric programming model of Signal/Collect exploits the fact that most graph algorithms can be expressed in two operations on a vertex. First, a vertex *signals* its state to neighboring vertices along the edges. Second, a vertex *collects* the incoming signals and computes its new state based on them. Both the signal and the collect steps are executed repeatedly for each vertex until a termination condition is fulfilled. In a synchronous execution, all the vertices are at the same time either in the signal phase or in the collect phase. An asynchronous execution overrides this restriction and executes the signal and collect steps for each vertex independently. The Signal/Collect framework is scalable because it is able to use additional resources to speed up the execution of algorithms. Parallelism is exploited by distributing the vertices of a graph to multiple workers, which then process their assigned vertices in parallel. Distributed workers running on different compute nodes are also supported [Freitas, 2011].

The most recent version of the Signal/Collect framework[1] distributes the vertices to the workers by using a simple function mapping vertex identifiers to worker identifiers. The resulting sets of vertices, each of which is assigned to one worker, are balanced and random. The sets are balanced because the number of vertices in each set is approximately equal. They are random because a vertex is assigned to a specific set by hashing its identifier and not using any additional information. For example, the edges interconnecting a vertex with other vertices are not considered when assigning a vertex to a set. As a result, there might be few edges between vertices in the same set and a lot of edges between vertices in different sets. Because each set is assigned to one worker and the signals sent by the vertices travel along these edges, the communication time and in the end the runtime of the computation are increased. This effect is amplified in a distributed environment, for example in a cluster or in the cloud, where the network latency is high, the network bandwidth is low, and the data to be sent to other compute nodes needs to be serialized and afterwards deserialized. It is therefore reasonable to

[1] The most recent version of Signal/Collect can be found at *https://github.com/uzh/signal-collect*. At the moment of this writing, the identifier of the most recent revision is `5fd6cec5a51296986e41f252d27408b07f4e8149`.

assume that more sophisticated approaches to distributing the vertices to the workers would improve the runtime performance of the Signal/Collect framework by reducing communication between different workers and compute nodes.

Two solutions are apparent. The first consists in replicating vertices on multiple workers. When a vertex is present on a worker due to replication, the need to send a signal to the original worker of the vertex is eliminated. Instead, the signal can be sent locally to the replicated vertex. In the literature, replication-based approaches are explored in the context of graph query processing systems, for example in [Yang et al., 2012]. However, this solution has several drawbacks. The most severe one is that the states of the vertices are updated very frequently in Signal/Collect. Although replication might eliminate the need to send signals to remote workers, the consistency between replicated vertices needs to be maintained, which requires communication between the workers. Furthermore, vertex replication will increase the memory consumption of the system. Because Signal/Collect is designed to also handle very large graphs, this is a significant drawback. Out of these two reasons it can be concluded that a replication-based approach is not well-suited to improve the runtime performance of Signal/Collect and is not considered further in this thesis.

The other solution lies in distributing the vertices to the workers in a more sophisticated way. Instead of distributing the vertices by hashing their identifiers, the edges between the vertices could be taken into account to minimize interconnections between vertices on different workers. The underlying theoretical problem is that of graph partitioning. The problem of graph partitioning consists in finding a specified number of distinct subsets of the set of vertices of a graph. Usually, these subsets have to be of approximately equal cardinality and the number of edges between the different subsets has to be minimized [Bichot and Siarry, 2011]. Graph partitioning is a well-researched field and is surveyed in Chapter 2. However, there are significant differences between graph partitioning as discussed in the literature — called *classical graph partitioning* hereafter — and the problem of distributing vertices to workers in Signal/Collect. First, graphs in Signal/Collect might be very large and global knowledge, which is centralized knowledge about the whole graph, might not be available due to the graph's size and the distributed nature of the system. In contrast, algorithms for classical graph partitioning very often assume a global view on the graph [Rahimian et al., 2013]. Second, graphs in Signal/Collect are dynamic. The structure of a graph might change over time by adding or removing vertices and edges. They are also dynamic in the sense that during an execution, certain edges might not transport any signals whereas others might be active in each step. Furthermore, this messaging patterns might change over time. Most algorithms for classical graph partitioning do not deal with such dynamic graphs. Third, the primary goal of graph partitioning in the context of Signal/Collect is not to find a perfect partitioning, which might take a lot of time, but to improve the runtime performance of the system. From this discussion, it can be concluded that most of the classical graph partitioning algorithms are not appropriate to solve the present problem.

Figure 1.1 shows that in graph processing systems like Signal/Collect, the partitioning of the graph can happen before, during, or after the computation. Based on these phases, we differentiate between *static* and *dynamic* graph partitioning. This terminology is also

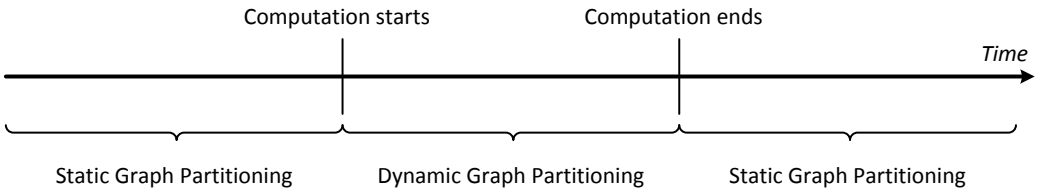

Figure 1.1: Static and dynamic graph partitioning

used in related literature, for example in [Salihoglu and Widom, 2012].

Static graph partitioning is executed before a computation is started or after it is ended. Partitioning the graph after a computation is ended is reasonable in case the graph is executed again with the same or a different algorithm. Static partitioning is a pre- or postprocessing step to the execution of a graph and does not modify the partitions during the computation. It is straightforward to integrate into the system, because not many changes to the core of Signal/Collect are necessary. Furthermore, numerous methods and established tools are available to statically partition a graph, e.g. METIS [Karypis and Kumar, 1998]. However, static graph partitioning has two significant drawbacks. It increases the time until the graph is loaded and the computation can be started. Also, static graph partitioning is by definition not able to react to changes in the graph during the computation.

On the other hand, dynamic graph partitioning runs after the computation is started and before it is ended. The preeminent advantage of this approach is that it is able to adjust the partitions continuously during the computation. Compared to the static approach, it is, however, more difficult to integrate into the system as it affects the core of the Signal/Collect system and cannot be implemented as a pre- or postprocessing step to the execution of the graph.

Dynamic graph partitioning can be further categorized into *serial* and *parallel* partitioning. Figure 1.2 shows that in case of serial dynamic graph partitioning, the computation is paused to partition the graph. Parallel dynamic graph partitioning partitions the graph while the computation is running, as Figure 1.3 shows.

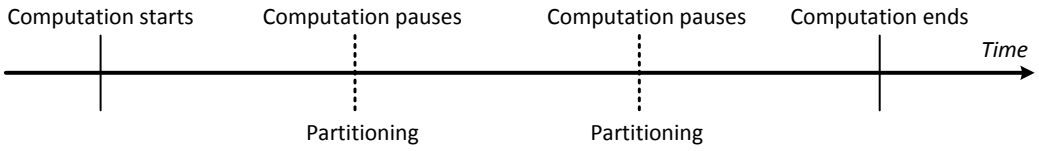

Figure 1.2: Serial dynamic graph partitioning

In this thesis, the Signal/Collect framework is extended to support both serial and parallel dynamic graph partitioning. The goal of the graph partitioning functionality is

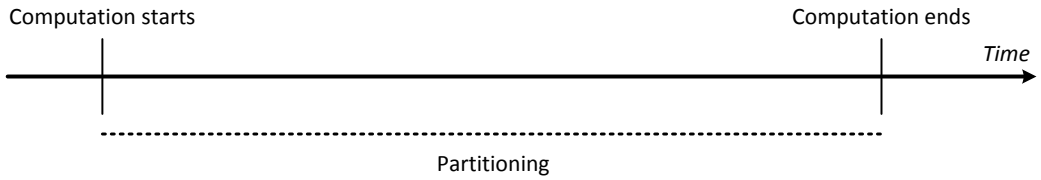

Figure 1.3: Parallel dynamic graph partitioning

the improvement of the runtime performance of Signal/Collect.

The structure of this thesis is as follows. Chapter 2 presents the related work. In Chapter 3, the design of the partitioning functionality and its necessary prerequisites are explained. Different partitioning algorithms are described in Chapter 4. In Chapter 5, the evaluation of the approach is presented and the results are discussed in Chapter 6. In Chapter 7, ideas for future work are described. Finally, the thesis closes with a summary in Chapter 8.

2

Related Work

This chapter presents the related work originating from different areas. The underlying theoretical problem of this thesis is that of *graph partitioning*. Methods for graph partitioning are used for *load balancing* in distributed systems. *Graph databases* store large graphs and need to distribute the vertices and edges to different compute nodes. Finally, the largest part of the related work can be found in the area of *graph processing systems*. All of these topics are surveyed in the following sections.

Graph Partitioning Graph partitioning has been a widely researched subject for decades and an accordingly large amount of literature exists. An overview on the problem, proposed algorithms, different fields of application, and available software packages can be found in [Bichot and Siarry, 2011]. [Fjällström, 1998] is an older, but more compact and focused survey. Despite the large amount of literature in the area of graph partitioning, suitable algorithms for the present problem are rare, as discussed in the previous chapter. Decentralized algorithms for graph partitioning can be found in [Gehweiler and Meyerhenke, 2010], [Rahimian et al., 2013], and [Ramaswamy et al., 2005]. Their need for extensive communication between vertices, and therefore workers and compute nodes, makes them unusable for the present problem. A dynamic graph partitioning extension for Signal/Collect should not interfere with the execution of the actual algorithm, because then the reduction of the runtime achieved by the partitioning could be eliminated by the time needed to process the additional messages and calculate the partitioning.

A special class of graph partitioning algorithms are the *streaming graph partitioning algorithms*, which are investigated in the context of graph processing systems. The basic idea is to determine the responsible worker for each vertex according to simple heuristics while loading the graph. Two recent papers that investigate such algorithms are [Stanton and Kliot, 2012] and [Tsourakakis et al., 2012]. However, streaming graph partitioning is a form of static graph partitioning, which is not investigated in the present thesis.

Whereas in graph partitioning the number of partitions is known beforehand, *graph clustering* methods try to find communities in graphs without being restricted by a specified number of communities [Fortunato, 2010]. Due to this fact, graph clustering methods are not well-suited for the present problem in which the number of workers, each of which stores exactly one partition, is specified before the graph is loaded.

Load Balancing The goal of load balancing in distributed systems is to minimize communication cost between compute nodes while balancing workload between them. A survey can be found in [Devine et al., 2006]. Modelled as a graph, this problem can be solved by applying graph partitioning algorithms. These methods are discussed in the previous section and no other significant contribution from the field of load balancing to the present problem has been found.

Graph Databases Graph database systems are database systems specialized in storing graphs. In case of large graphs, these database systems have to distribute the vertices and edges to multiple compute nodes. Furthermore, the stored graph might be modified over time, because edges and vertices are added or removed. Therefore, graph databases face similar challenges like Signal/Collect. However, Averbuch and Neumann [Averbuch and Neumann, 2013] mention that existing literature about partitioning graph databases is scarce. In their thesis, they extend the Neo4j database[1] with the already mentioned distributed graph partitioning algorithm presented in [Gehweiler and Meyerhenke, 2010].

Graph Processing Systems In recent years, many graph processing systems have been developed and discussed in the literature. In this section, they are surveyed with emphasis on their support for graph partitioning.

The Graph Processing System (GPS) [Salihoglu and Widom, 2012] is a system for the execution of algorithms on large graphs. GPS supports static and dynamic graph partitioning mechanisms. The static graph partitioning is customizable. The authors use both random partitioning and the graph partitioning tool METIS in their experiments. The dynamic graph partitioning mechanism, which is not customizable, works as follows. Each worker constructs a set of potentially transferable vertices for each other worker. The selection of vertices is based on the number of messages a certain vertex sends to vertices of the other worker. Afterwards, each pair of workers compares the sizes of the two sets. The smaller number of vertices is then exchanged to maintain a balance between the number of vertices on each worker. The dynamic graph partitioning mechanism leads, however, only to modest improvements in the runtime performance of GPS.

Khayyat et al. developed the graph processing system Mizan. This system supports static and dynamic graph partitioning methods. In [Khayyat et al., 2012], the authors describe Mizan as a layer between a graph processing system and the physical computing infrastructure. Mizan offers advanced static graph partitioning mechanisms involving the usage of METIS, vertex replication, and message passing in a virtual overlay ring. In [Khayyat et al., 2013], Mizan is enhanced by dynamic load balancing of communication and computation across all compute nodes. In contrast to their first paper, the authors herein describe Mizan as a graph processing system. Each worker records for each of its vertices the number of incoming messages, the number of outgoing messages, and the processing time. Workers exchange summaries of these statistics among each other. After each superstep, a vertex migration plan is generated on each worker based on these

[1] *http://www.neo4j.org*

statistics. After the next superstep, the migration plans are executed and the vertices are transferred. Interestingly, the Mizan system does not take the graph structure into account when moving vertices, as the decision is based on load balancing considerations only. The authors report large performance gains for highly dynamic workloads.

Shang et al. [Shang and Yu, 2013] built a graph processing system supporting dynamic graph partitioning on top of Apache Hama[2]. Each worker decides by itself how many, and which vertices it moves to the other workers. The selection of vertices is based on the following rules. First, overloaded workers send vertices to underloaded workers. Second, a vertex is moved to the worker which communicates most intensively with the vertex. In contrast to GPS, the present system considers both incoming and outgoing messages to a vertex. Third, a vertex is not moved frequently, but only if a significant improvement is to be expected. This means that the target worker should communicate much more intensively with the vertex to be moved than the source worker. The authors report a slight improvement of the runtime performance when using dynamic graph partitioning.

Grace [Prabhakaran et al., 2012] is a graph management system built for real-time queries and iterative computations on multicore systems. With hash-based partitioning, heuristics-based partitioning, and spectral partitioning it supports different algorithms for static graph partitioning. Grace also offers dynamic load balancing between threads, but, as a shared memory system, it differs significantly from Signal/Collect and the previous three systems.

Unlike the approaches discussed so far, most of the related work discusses static graph partitioning only.

Pregel [Malewicz et al., 2010] is a system developed for large-scale graph processing at Google. In this framework, a graph is divided into partitions which in turn are assigned to workers. The default partitioning function is a hash function on the identifiers of the vertices. This function can be replaced by the users of the framework and, for example, colocate vertices representing pages of the same web site in case of the Web graph on the same compute node. Thus, Pregel supports customizable static graph partitioning, but it does not offer dynamic graph partitioning. However, dynamic graph partitioning is mentioned as a challenge likely to be addressed.

Chen et al. [Chen et al., 2012] investigate graph partitioning for graph processing systems in cloud environments. Their system, Surfer, statically partitions the graph taking the network topology of the executing compute nodes into account. For example, partitions with a lot of edges to other partitions are assigned to compute nodes with a high bandwidth.

HipG [Krepska et al., 2011] offers customizable static graph partitioning using a hash function by default.

GraphLab [Low et al., 2012] uses a two-stage technique to implement static graph partitioning. First, the graph is partitioned by a customized partitioning, random partitioning, or a tool like METIS, whereas the number of partitions exceeds the number of available workers. Second, the partitions from the first step themselves form a graph

[2] *http://hama.apache.org*

which is then again partitioned and these partitions are finally assigned to the workers.

GraphInc [Cai et al., 2012] is a graph processing system supporting real-time graph mining built on top of Apache Giraph[3]. The user specifies vertex-centrix algorithms, which are then converted into incremental algorithms suitable for real-time processing by the system. GraphInc partitions the vertices randomly.

Trinity [Shao et al., 2012] is a distributed database and computation platform that supports online query processing and offline graph analytics. For static graph partitioning, it partitions vertices by using a hash function.

Horton [Sarwat et al., 2012] is a query execution engine for large graphs. Although the graphs are partitioned into several partitions, the mechanisms used for this process are not discussed more in depth. The authors propose the usage of graph partitioning algorithms or hash based partitioning.

Ho et al. [Ho et al., 2012] describe a two-layered system consisting of a graph processing layer and an underlying graph storage layer. The system uses METIS to statically partition the graph. When a vertex is added later, it is not immediately inserted into a partition but first cached and observed with regard to its communication patterns. Based on these observations, it is then assigned to a suitable partition.

Najork [Najork, 2009] describes the Scalable Hyperlink Store, which is a distributed in-memory database for storing large parts of the web graph. The web graph is statically partitioned by using portions of the URIs.

In [deLorimier, 2013], static graph partitioning in the form of reducing edges between partitions and balancing workload is discussed as an optimization technique to a graph processing system.

Redekopp et al. [Redekopp et al., 2013] discuss graph processing in the cloud. Having built a system similar to Pregel, they also investigate the effect of graph partitioning to the runtime performance of the system. They found out that static graph partitioning methods might decrease the runtime of algorithms, because the barrier in synchronous execution lets the system wait for the slowest worker. They do not discuss dynamic graph partitioning, however.

In contrast to other systems, the PowerGraph system [Gonzalez et al., 2012] assigns each edge to one partition and allows vertices to span multiple compute nodes. Another system following this approach is [Xin et al., 2013] based on [Zaharia et al., 2010]. Both systems implement static graph partitioning only.

There are other graph processing systems that do not address the problem of partitioning at all. These include [Haller and Miller, 2011], [Bykov et al., 2011], and [Gregor and Lumsdaine, 2005]. The authors of LFGraph [Hoque and Gupta, 2013] claim to have eliminated the need for graph partitioning by leveraging techniques for reducing the communication between compute nodes.

Some of the graph processing systems are based on the MapReduce [Dean and Ghemawat, 2008] programming model. Expressing graph algorithms in this model is challenging and graph partitioning is one aspect that needs to be taken care of [Lin and Schatz, 2010].

[3] *http://giraph.apache.org*

GBASE [Kang et al., 2011] is a graph management and mining system built on top of Apache Hadoop[4], a framework that implements the MapReduce programming model. GBASE allows any graph partitioning method to be used for the initial static graph partitioning.

PrIter [Zhang et al., 2011] uses and extends iMapReduce [Zhang et al., 2012]. Whereas the latter does not address graph partitioning, PrIter is able to automatically partition a graph or to use a method supplied by the user of the system.

Choi et al. [Choi et al., 2012] discuss graph partitioning by using a semi-clustering algorithm specifically for the execution of the PageRank algorithm on Apache Hadoop.

MapReduce based graph processing systems that do not discuss partitioning at all include PEGASUS [Kang et al., 2009], HaLoop [Bu et al., 2010], and Twister [Ekanayake et al., 2010].

The importance of the graph partitioning problem can be reduced by implementing the graph processing system as a shared memory architecture. Such approaches are described in [Shun and Blelloch, 2013], [Kyrola et al., 2012], [Low et al., 2010], and [Wang et al., 2013].

Graph processing systems that are not covered by literature include Apache Giraph, Apache Hama, GoldenOrb[5], JPregel[6], Phoebus[7], and Bagel[8].

Other Related Work Fard et al. [Fard et al., 2012] discuss issues regarding time evolving graphs. They notice that three aspects should be considered with regard to graph partitioning of time evolving graphs. The first aspect is the nature of the computation. While some computations like PageRank might benefit from minimizing the communication between compute nodes, other algorithms like Single Source Shortest Path (SSSP) will not. The second aspect is the dynamic structure of the graph. In time evolving graphs, vertices and edges are added and deleted over time. To deal with this issue, dynamic graph partitioning might be applied. However, the authors explicitly point out the expensiveness of this operation. The third aspect is the way the compute nodes are configured. The authors advise to generate more partitions than available compute nodes.

[4] *http://hadoop.apache.org*
[5] *http://goldenorbos.org*
[6] *http://kowshik.github.io/JPregel*
[7] *https://github.com/xslogic/phoebus*
[8] *https://github.com/mesos/spark/wiki/Bagel-Programming-Guide*

3

Design

In this chapter, the design of the dynamic graph partitioning functionality in Signal/Collect is described. First of all, Section 3.1 gives an overview on the existing Signal/Collect architecture, necessary to understand the subsequent sections. Section 3.2 describes how vertices can be localized. The displacement of vertices between workers is explained in Section 3.3. After having introduced the prerequisites, the dynamic graph partitioning functionality is finally presented in Section 3.4.

3.1 Signal/Collect

The Signal/Collect framework can be logically split up into several parts. The following list enumerates its most important components:

- The *driver* initializes the system and offers an interface to the client of the Signal/Collect framework. Using this interface, the graph can be constructed and modified. It also offers methods to execute the graph and to shut it down afterwards.
- The *coordinator* is responsible for observing the *workers* and the *node controllers*. It collects status messages from those entities, which are used to determine whether the execution of an algorithm has finished.
- Each *node controller* controls the lifecycle of a set of workers. The workers are created and shut down by their node controllers.
- Each *worker* processes a set of vertices.

All of these components are present in each *execution* of the framework. An execution consists of one or more *instances* of the Signal/Collect framework. In case of a *local execution*, there is only one instance comprising the driver, the coordinator, one node controller, and at least one worker. This case is illustrated in Figure 3.1. In contrast, a *distributed execution* consists of several instances. There is one *master instance*, which accommodates the driver and the coordinator, and at least one *normal instance* consisting of a node controller and at least one worker. Figure 3.2 shows a distributed execution consisting of several instances each running on a different compute node. One of these

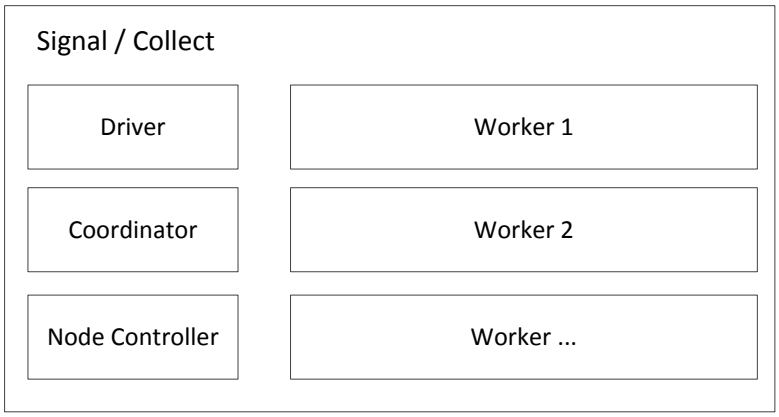

Figure 3.1: Local Signal/Collect execution

instances is the master instance, as shown in Figure 3.3. The other instances are normal instances, as illustrated by Figure 3.4. Although it would be possible to run several instances on one compute node, the usual case is to run one instance on one compute node. Therefore, we use the terms *instance* and *compute node* interchangeably in the following.

In general, all of the listed entities can communicate with each other. Notable communication channels are between the workers, which exchange signals for their vertices, and between the coordinator, the workers, and the node controllers, which exchange status messages.

The subsequent sections describe how the existing architecture of Signal/Collect is extended to support dynamic graph partitioning.

3.2 Vertex Localization

To send a message to a particular vertex, a worker needs to know the identifier of the worker responsible for that vertex. So far, Signal/Collect has used a simple function to calculate the worker responsible for a certain vertex. This function is first used when a vertex is added to the graph and the system needs to assign it to a particular worker. The same function is called afterwards, whenever the worker responsible for that vertex is looked up, for example when a vertex sends a message to another vertex.

Concretely, Signal/Collect uses a vertex localization function m, which maps a vertex identifier to a worker identifier by calculating a hash value of the vertex identifier and relating it to the total number of workers in the system. Let $\mathbb{W} = \{0, 1, 2, ...\}$ be the set of worker identifiers. Given a hash function $h : \mathbb{V} \to \mathbb{Z}$, mapping vertex identifiers in $\mathbb{V}$

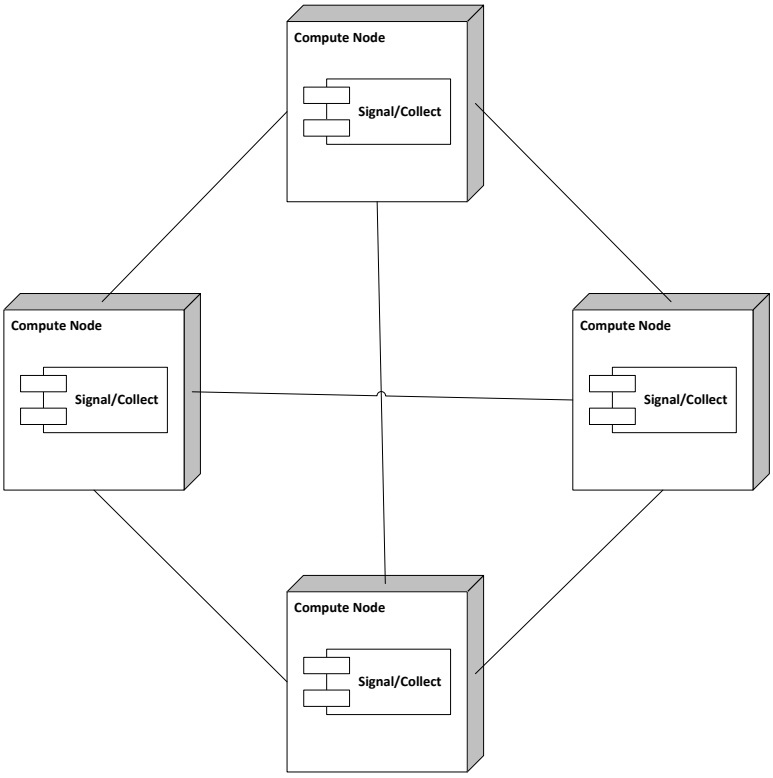

Figure 3.2: UML deployment diagram of a distributed Signal/Collect execution

to integers in $\mathbb{Z}$, m can be defined as follows:

$$\begin{aligned} m :& \mathbb{V} \to \mathbb{W} \\ & v \mapsto |h(v) \mod |\mathbb{W}|| \end{aligned} \tag{3.1}$$

To support dynamic graph partitioning — or more general, arbitrary assignments of vertices to workers — this approach is not sufficient. Although m can be implemented efficiently, it prohibits arbitrary assignments by restricting the set of possible workers for a certain vertex to a single one.

Arbitrary assignments of vertices to workers could be implemented by storing the vertex identifier and its corresponding worker identifier in a table. However, this approach does not scale to large graphs with millions of vertices. A different solution is to extend the existing lookup function by a preceding table. That table does not store the location for each vertex, but only for those vertices which are not located on their *default*

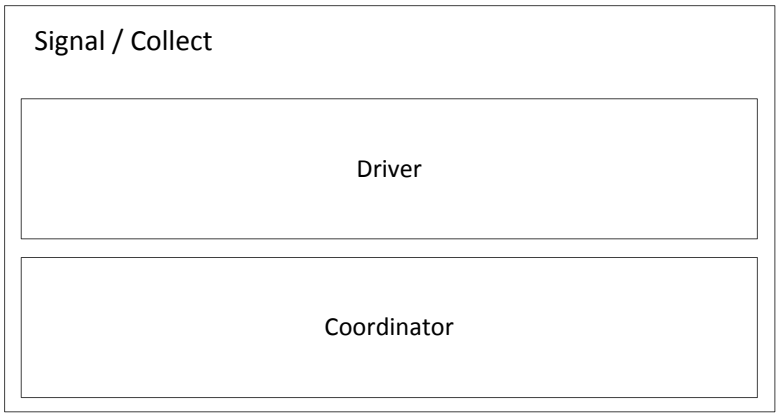

Figure 3.3: Signal/Collect master instance in a distributed execution

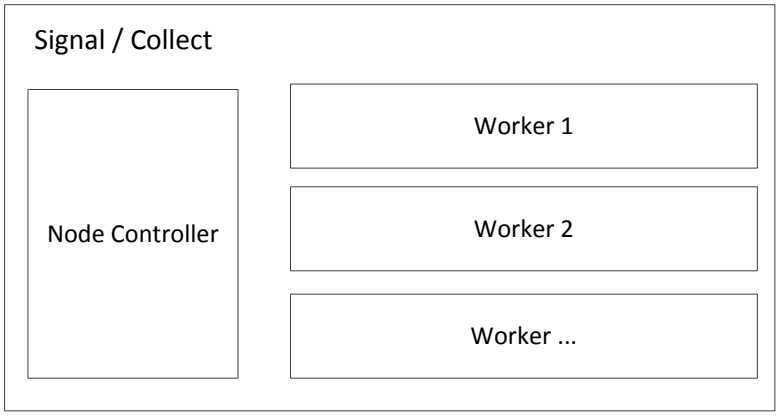

Figure 3.4: Signal/Collect normal instance in a distributed execution

worker. The default worker is given by the existing lookup function. Thus, the table stores exceptional placements of vertices and is therefore called *exception table*. This approach uses less memory than storing the location for all vertices. However, even this solution is likely to encounter performance issues when the number of vertices which are not located on their default workers is very large. Algorithm 1 shows a concrete implementation of this lookup procedure.

To integrate this approach into the Signal/Collect architecture, it is sufficient to store one exception table on each instance of the Signal/Collect framework. Each worker of a certain instance can then access this table, whereas the management of the table — updating and removing entries — is reserved to the node controller of that instance.

Algorithm 1 Returns the identifier of the worker responsible for a vertex identifier.

```
function WorkerIdentifier(exceptionTable, numberOfWorkers, vertexIdentifier)
    if exceptionTable.contains(vertexIdentifier) then
        return exceptionTable.get(vertexIdentifier)
    else
        return |vertexIdentifier.hashCode()  mod numberOfWorkers|
    end if
end function
```

3.3 Vertex Displacement

In order to support dynamic graph partitioning, the system has to offer a mechanism to move vertices between workers. When a vertex is moved from the worker W_0 to the worker W_1, W_0 is called the *source worker* and W_1 is called the *target worker*. The vertex displacement functionality is implemented in terms of the two worker methods *moveVertex* and *takeVertex*.

The first step in the vertex displacement process is a call to the *moveVertex* method on the source worker. It moves the vertex to the target worker and removes it from the source worker. Algorithm 2 shows this method. The *moveVertex* method accepts a vertex and a target worker. First, an asynchronous request to take the vertex is sent to the target worker. In a second step, the vertex is removed from the source worker. Finally, the source worker's exception table is updated to store the new location of the worker. The last step causes any message for the vertex just removed to be forwarded to the target worker. In case the target worker has already fulfilled the asynchronous request to add the vertex, it accepts the message. On the other hand, if the vertex is not yet available on the target worker, the message is routed back to the source worker. There, this cycle starts again until the target worker finally adds the vertex to its store and accepts corresponding messages.

Algorithm 2 Moves the vertex to the target worker.

```
function moveVertex(vertex, targetWorker)
    targetWorker.takeVertex(vertex)
    removeVertex(vertex)
    nodeController.setVertexLocation(vertex.identifier, targetWorker.identifier)
end function
```

The *takeVertex* method is called on the target worker. It adds a vertex to the target worker and updates all exception tables in the system. An implementation of this method is shown in Algorithm 3.

First, the vertex is added to the target worker. Second, all node controllers are sent a request to update the location of the vertex to its new worker. After the execution of the *takeVertex* method, the worker having added the vertex accepts corresponding messages and the other workers send messages to the vertex directly to its new worker, because

Algorithm 3 Adds the vertex to the worker.

```
function TAKEVERTEX(vertex)
    addVertex(vertex)
    nodeControllers.setVertexLocation(vertex.identifier, workerIdentifer)
end function
```

all the exception tables in the system have been updated to store the new location of the vertex.

Given these two methods, vertices can be moved from one worker to the other.

3.4 Dynamic Graph Partitioning

After having presented all necessary prerequisites, this section describes the design of the dynamic graph partitioning functionality in detail. Architecture, protocol, scheduling, and partitioner aspects are discussed separately.

3.4.1 Architecture

Given the architecture of Signal/Collect as described in Section 3.1, there are several possible approaches to integrate dynamic graph partitioning. Three different models have been identified and are discussed in the following subsections.

Centralized Model

In the centralized model, there is one entity in the system that is responsible for partitioning the graph. The first advantage of this model is that the partitioning process is easier to reason about and to control, because it is centralized. Furthermore, the centralized partitioner might have global knowledge of the graph and use existing partitioning tools like METIS. However, the main disadvantage of the centralized model is that the data necessary to find suitable partitions needs to be transferred from the workers to the central partitioner. In the worst case, these transfers happen frequently and the amount of data is large. This significant disadvantage disqualifies the centralized model as a suitable solution to the partitioning problem in Signal/Collect. No other graph processing system implements the centralized model for dynamic graph partitioning, either.

Worker Model

In the worker model, the workers are responsible for the partitioning process. Each worker has access to its own internal state and calculates vertex displacement requests based on this information. Besides the requests, workers might periodically send each other status messages. In contrast to the centralized model, not much information is sent around. The worker model has the disadvantage that the computation, which takes place in the workers, is interrupted when the partitioning occurs. Furthermore, partitioning

is an additional responsibility for the workers. Therefore, the worker model is not an optimal solution for the Signal/Collect framework. Nevertheless, this model is used in the approaches described in [Khayyat et al., 2013], [Salihoglu and Widom, 2012], and [Shang and Yu, 2013].

Node Controller Model

In the node controller model, the node controllers are responsible for partitioning the graph. The node controllers have access to the internal states of the workers for which they are responsible. To gain information about remote workers and nodes, the node controllers can send each other summary messages. Therefore, the amount of information sent around is limited compared to the centralized model. In contrast to the worker model, the node controller model does not interrupt the workers to calculate the partitioning. Furthermore, node controllers are otherwise idle as their only function in the current architecture of Signal/Collect is to create workers. It can be concluded that the node controller model is well-suited to integrate dynamic graph partitioning into the Signal/Collect framework. So far, no other system has implemented the node controller model for dynamic graph partitioning.

3.4.2 Protocol

The partitioning mechanism uses a one-way protocol in which the *vertex request* is the only message type. A vertex request contains a vertex identifier, the identifier of the source worker, the identifier of the target worker, and a weight. A vertex request is either a *vertex pull request* or a *vertex push request*. In case of a pull request, the message is a request to move the vertex given by its identifier from the target worker to the source worker. In case of a push request, the message is a request to move the vertex given by its identifier from the source worker worker to the target worker. In both cases, the weight can be used to prioritize multiple vertex requests. Vertex requests are sent between node controllers and the receiving node controller decides whether a vertex request should be executed. Executing a vertex request means that the displacement of the vertex denoted in the request is initiated.

In contrast to the other systems supporting dynamic graph partitioning, the present framework is the first to support both pulling and pushing vertices. In the other known solutions, vertices are always pushed from one worker to the other. It is unclear whether the GPS system [Salihoglu and Widom, 2012] implements a one-way or multiple-ways protocol. The dynamic graph partitioning mechanisms discussed in [Khayyat et al., 2013] and [Shang and Yu, 2013] do not know the concept of a request, because in those systems vertices are sent directly without any form of negotiation at all.

Execution Mode	Scheduling	
	Serial	Parallel
Synchronous	X	
Pure Asynchronous	X	X
Optimized Asynchronous	X	X
Continuous Asynchronous		X
Interactive		

Table 3.1: Signal/Collect execution modes and supported scheduling methods

3.4.3 Scheduling

To use dynamic graph partitioning, the client of the Signal/Collect framework needs to specify a *scheduler*, which decides when the next partitioning of the graph should happen. The framework currently supports scheduling by time, although other criterias, e.g. the number of signal and collect steps, could be supported, too. Scheduling the partitioning by time is a new concept, as the other graph processing systems implementing dynamic graph partitioning partition the graph after a certain number of computation steps. Corresponding to the notion of serial and parallel dynamic graph partitioning as presented in Chapter 1, serial and parallel schedulers were implemented.

In the case of serial scheduling, the partitioning process is controlled by the driver. It periodically calls the scheduler to check whether a partitioning is necessary. When the scheduler demands a partitioning, the computation is paused, the node controllers are requested to partition the graph, and then the computation is restarted.

In the case of parallel scheduling, the scheduler does not reside on the driver, but each node controller has its own scheduler. The node controllers periodically call their schedulers to check whether a partitioning is necessary and start it as soon as demanded. Each node controller partitions the graph independently.

Table 3.1 shows which scheduling methods are supported by the different execution modes. The interactive execution mode does not support any scheduling method, as it exists mainly for visualization purposes. Parallel scheduling has not been implemented for the synchronous execution mode. In contrast, serial scheduling has not been implemented for the continuous asynchronous mode, which is a special execution mode for computations that do not terminate, e.g. query processing systems. However, the other asynchronous execution modes support both serial and parallel schedulers.

3.4.4 Partitioner

When executing Signal/Collect with the dynamic graph partitioning functionality, each node controller has its own instance of the partitioner. A partitioner needs to solve two tasks. First, it has to decide which vertices to send requests for. Second, it needs to process the received requests and execute the desired ones. The implemented framework allows a wide variety of different algorithms. Several possible ones are discussed and evaluated in the next chapter.

4

Partitioning Algorithms

After having described the framework for dynamic graph partitioning in Signal/Collect, this chapter presents several partitioning algorithms and discusses the best performing candidate in detail.

4.1 Goal

The goal of dynamic graph partitioning in Signal/Collect is the improvement of the runtime performance. This goal can be achieved by reducing signals sent between compute nodes, called *remote signals*, and balancing the amount of work, called *load*, among the workers. However, these two objectives are conflicting. For example, remote signals could be eliminated completely by assigning all vertices to one worker. But that overloaded worker would need a lot of time to process all its vertices while the other workers would be idle. Of course, memory limitations might also make this approach impossible. In contrast, an acceptable load balancing could be obtained by assigning all vertices in a round-robin fashion to the workers. This strategy, however, might increase the number of remote signals, because it does not take the edges between the vertices into account by arbitrarily distributing the vertices to the workers. Any partitioning algorithm aiming to reduce the execution time of algorithms in Signal/Collect has to take both objectives, reducing remote signals as well as load balancing, into account.

4.2 Approaches

To select a dynamic graph partitioning approach, it is sufficient to look at the reduction of the number of remote signals a particular algorithm achieves. Any approach can later be extended with load balancing.

The number of remote signals can be reduced by assigning vertices which communicate with each other to the same compute node. The choice of a particular worker on that compute node is largely irrelevant, for workers on the same compute node can communicate very efficiently with each other. To determine the vertices communicating with each other, the edges of each vertex could be investigated. However, this method has two drawbacks. First, it does not consider the actual number of signals sent along

a certain edge. There might be edges which do not transport any signals and others which send a lot of signals. Second, in Signal/Collect vertices can send messages to other vertices even if there is no actual edge connecting them. Both of these drawbacks can be circumvented by analyzing not the edges, but only the actual number of signals sent between vertices. To implement this approach, additional data structures keeping track of these numbers have to be integrated into the system. Therefore, the drawback of this approach is an increase of memory consumption.

Four partitioning approaches based on the idea of analyzing the number of signals sent between vertices have been developped:

Push1 This partitioner pushes each vertex to the worker it sent most messages to, if that particular worker is not on the same compute node as the vertex.

Push2 This partitioner pushes each vertex to the worker to which it sent most messages to on the compute node to which it sent most messages to.

Pull1 Using this partitioner, vertices are moved to the worker which sent most messages to this vertex, if that particular worker is not on the same compute node as the vertex.

Pull2 Using this partitioner, vertices are moved to the worker which sent most messages to this vertex on the compute node which sent most messages to this vertex.

In the related work, [Salihoglu and Widom, 2012], [Shang and Yu, 2013], and [Khayyat et al., 2013] partition the graph according to the number of messages between vertices, too. [Khayyat et al., 2013] attempts to balance the number of received and sent messages for each worker. The approach described in [Salihoglu and Widom, 2012] pushes vertices to the workers to which they sent most messages to. This idea is implemented in the Push1 and Push2 approaches. [Shang and Yu, 2013] also consider the number of received messages by a vertex, which we do not. The number of received signals cannot be used, because in Signal/Collect a received signal does not necessarily contain the identifier of the sender.

4.3 Preevaluation

The purpose of the preevaluation is to select an approach which will then be extended by load balancing and fully evaluated in the next chapter. To evaluate the four approaches, the following procedure is used:

1. Execute an algorithm.
2. Measure the number of sent remote signals.
3. Partition the graph according to the selected approach.
4. Execute the algorithm again.

	Number of Remote Signals		
Partitioner	Before Partitioning	After Partitioning	Ratio
Push1	414739 (20916)	265394 (4871)	0.64
Push2	418333 (19078)	265370 (5597)	0.63
Pull1	424217 (26085)	295870 (9091)	0.70
Pull2	423813 (21140)	279974 (8152)	0.66

Table 4.1: Median number of remote signals and median absolute deviation in parenthesis when 4-coloring the vertices of a 100x100 grid graph 50 times using 3 compute nodes each with 1 worker

5. Measure the number of sent remote signals.

The best performing approach is the one which achieves the largest reduction of remote signals between the two executions of the algorithm.

All approaches are evaluated using two evaluation cases.[1] The first evaluation case is an algorithm for vertex coloring, which assigns each vertex of a graph a color such that no neighboring vertex has the same color. This algorithm is executed 50 times before partitioning and 50 times after partitioning on a 100×100 grid using 4 colors. Signal/Collect runs on 3 compute nodes each with 1 worker.[2]

The results of this evaluation can be seen in Table 4.1. In comparison, the two push-based approaches perform well. Pull2 achieves almost the same reduction as the two push-based approaches, but the Pull1 approach is distant fourth in the ranking.

The second evaluation case is TripleRush[3], which is a store for Resource Description Framework (RDF)[4] triples built on top of Signal/Collect. To evaluate the four approaches, a subset of the Lubm160 benchmark data set [Zeng et al., 2013] is used and each of the 7 queries of the benchmark is executed 10 times before and after the partitioning. Signal/Collect runs on 4 compute nodes each with 24 workers.[5]

The results of this evaluation can be seen in Table 4.2. It is obvious that the pull-based partitioners perform worse than their push-based counterparts. As in the vertex coloring experiment, Pull2 performs significantly better than Pull1. Push1 seems to

[1] For details on the evaluation environment and the evaluation cases, the reader is referred to chapter 5. The raw evaluation results of all evaluations presented in this chapter are provided on the accompanying CD-ROM.

[2] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `af03a58ad76b0aa9a57298eb650a87746bff4f8d` for the `signal-collect` project and `c9ccc66e0af6cdc83f859dfe77c053bd26d6986e` for the `mte` project. The evaluation is contained in the file `VertexColoringBaselineEvaluation.scala` in the `mte` project on the accompanying CD-ROM.

[3] *https://github.com/uzh/triplerush*

[4] *http://www.w3.org/RDF*

[5] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `af03a58ad76b0aa9a57298eb650a87746bff4f8d` for the `signal-collect` project and `2b8371f9cab7ffccdd3b56e9b4b8f2f96ce92797` for the `triplerush` project. The evaluation is contained in the file `Lubm160BaselineEvaluation.scala` in the `triplerush` project on the accompanying CD-ROM.

No. Vertices	32323			299142		
Partitioner	Before P.	After P.	Ratio	Before P.	After P.	Ratio
Push1	7180	5490	0.76	65510	54820	0.84
Push2	7180	5460	0.76	65510	58200	0.89
Pull1	7180	7820	1.09	65510	69600	1.06
Pull2	7180	7330	1.02	65510	63850	0.97

No. Vertices	1407199			2705640		
Partitioner	Before P.	After P.	Ratio	Before P.	After P.	Ratio
Push1	306280	284040	0.93	610620	560150	0.92
Push2	306280	271150	0.89	610620	549940	0.90
Pull1	306280	317170	1.04	610620	630680	1.03
Pull2	306280	289050	0.94	610620	572260	0.94

Table 4.2: Number of remote signals when executing each of the 7 queries of the Lubm160 benchmark 10 times on different subsets — 1, 10, 50, and 100 splits — of the benchmark's full data set on 4 compute nodes each with 24 workers

perform better than Push2 when the number of vertices is small, but Push2 improves its performance when the number of vertices increases.

Based on the collected data, it can be said that the push-based approaches perform better than their pull-based counterparts. This fact is explicable by considering that a pull-based approach pulls a vertex based on the number of messages it sent to it. However, it is not able to estimate the messaging patterns of that vertex. Therefore, a pull-based approach has no possibility to prevent pulling a vertex which itself sends a lot of messages to other vertices.

Although the pull-based approaches can be ignored due to their bad results in the experiments, it is difficult to draw a well-funded decision between the Push1 and Push2 approaches. In the evaluations, both approaches perform approximately equally well. However, Push2 most likely represents the idea of assigning vertices that communicate with each other to the same compute node. Based on this fact and the evaluation results, it is reasonable to assume that Push2 will perform equally well as the Push1 approach in most cases, and in some cases even better. Therefore, the Push2 approach is investigated in the rest of this thesis.

4.4 Push2

The basic idea of Push2 is to push each vertex to the compute node it sent most messages to. The purpose of this section is to discuss the algorithm in detail. First, the load balancing approach is described. The subsequent sections then discuss how Push2 solves the two fundamental tasks of any partitioning algorithm. First, it has to select vertices to send requests for. Second, it has to decide which of the received requests to execute.

	N_0		N_1		N_2	
Vertex	W_0	W_1	W_2	W_3	W_4	W_5
1	2	3	0	1	1	1
2	1	1	4	1	2	2
3	2	1	1	2	0	1
4	0	0	1	2	3	0

Table 4.3: Sample data showing the number of messages from the vertices $1-4$ to the workers W_{0-5} on the compute nodes N_{0-2}

4.4.1 Load Balancing

To balance load among the workers, different approaches are possible. A simple strategy is to aim for an approximately equal number of vertices on each worker. The underlying assumption of this approach is that the time needed to process a vertex is comparable for all vertices. Other approaches could consist of measuring for each vertex the time needed to process it, or estimating the time needed to process a vertex based on its number of edges.

The Push2 partitioner supports load balancing using the first strategy. It aims to keep the number of vertices on each worker in the range $[0.9a, 1.1a]$, where a is the average number of vertices per worker in the whole system.

4.4.2 Sending Vertex Requests

Each time the Push2 partitioner is called to send vertex requests, it examines a limited subset of each worker's vertices. The subset is limited in order to decrease the runtime of the Push2 algorithm. For each vertex of a certain subset, the Push2 partitioner checks whether there is a compute node to which the vertex sent more messages to than the current one. If this is the case, a vertex push request is sent to the worker to which the vertex sent most messages to on that compute node. The number of vertex requests to send is limited by two aspects. First, the load balancing scheme restricts the number of vertex requests to send as explained above. Second, the number of vertex requests to send is limited by a fixed number of allowed vertex requests equal for all workers of the system. The latter restriction is necessary to prevent that too many requests are sent in one partitioning step.

The decision procedure of Push2 is best illustrated using the sample data in Table 4.3. It is assumed that the partitioner is executed on node N_0 and all listed vertices are stored on worker W_0. Vertex 1 sends 5 messages to node N_0, 1 message to node N_1, and 2 messages to node N_2. Thus, this vertex is on the node it sent most messages to and no further action is taken. Vertex 2 sends 2 messages to node N_0, 5 messages to node N_1, and 4 messages to node N_2. To put this vertex to the compute node it sent most messages to, it has to be moved to node N_1. Therefore, a vertex request is sent to worker W_2, because this is the worker to which the vertex sent most messages to on the node to which the vertex sent most messages to. Vertex 3 is not moved, because it

sends 3 messages to node N_0, where it resides, and to node N_1. Therefore, there is no remote compute node to which it sent more messages to than to its current one. Vertex 4 might be moved to node N_1 or N_2, because it sent 3 messages to each of them and the algorithm is not deterministic in this regard.

4.4.3 Executing Vertex Requests

When processing the received vertex requests, the Push2 partitioner first calculates for each worker how many vertex requests it is allowed to execute, while still remaining within the limits given by the load balancing scheme. For example, given an average number of vertices per worker of 50, a worker owning 60 vertices is not allowed to execute any requests. In the same situation, a worker responsible for 40 vertices might execute up to 15 vertex requests.

After having determined the number of vertex requests allowed to be executed, the requests are handled in a first-come first-served fashion. Push2 afterwards removes all remaining vertex requests.

5

Evaluation

In this chapter, the Push2 partitioning algorithm is evaluated using two different evaluation cases.[1]

5.1 Environment

The evaluations were executed on the cluster of the Dynamic and Distributed Information Systems Group at the University of Zurich. The cluster consists of 12 machines each with 2 AMD Opteron™ 6174 processors and 66 GB RAM. The machines run a Linux 3.2.0-4 kernel and the IcedTea 2.1.7 64-bit Java virtual machine.

5.2 Vertex Coloring

The first evaluation case for dynamic graph partitioning is vertex coloring. The problem of vertex coloring consists of finding a color for each vertex of a graph such that no neighboring vertex has the same color. The algorithm to solve this problem works as follows.[2] Initially, all vertices of the graph have the same color. In the first signal step, all vertices signal their colors to all neighboring vertices. After that, in each collect step each vertex checks whether it has received a signal having the same value as its current color. If this is the case, it randomly decides to randomly choose a new color or to retain the old one. In the next signal step, it signals its color to its neighboring vertices. Otherwise, if no signal has the same value as its current color, the vertex does not send any signals in the next signal step.

This problem is an appropriate evaluation case, because it involves a lot of communication between the vertices and the messaging patterns change over time. For example, an area of the graph might be correctly colored while other areas are still being colored.

In the evaluation, 3 compute nodes each with 1 worker are used. The graph to be colored using 3 colors is a grid graph with the dimensions 10×10. The algorithm is executed 50 times using 4 different versions of Signal/Collect:

[1] The raw evaluation results of all evaluations presented in this chapter are provided on the accompanying CD-ROM.

[2] A concrete implementation of this algorithm can be found in the file `NaiveVertexColoring.scala` in the `mte` project on the accompanying CD-ROM.

- The default version of Signal/Collect.[3]
- The default version of Signal/Collect, but the vertices assigned to the workers according to a graph partitioning calculated by METIS.[4]
- Signal/Collect extended with dynamic graph partitioning and a parallel scheduler partitioning every 2 seconds.[5]
- Signal/Collect extended with dynamic graph partitioning and a serial scheduler partitioning every 2 seconds.[6]

It is to be expected that the default version of Signal/Collect with the default partitioning scheme performs worst. The best performance should result from the default version of Signal/Collect with the vertices assigned to the workers according to a graph partitioning calculated by METIS. This version should perform best, because its graph is partitioned optimally already at the start of the execution and no partitioning process runs in parallel. Regarding the versions extended with dynamic graph partitioning, the execution using the parallel scheduler should perform better than the one using the serial scheduler, because the serial scheduler interrupts the computation completely whereas the parallel scheduler executes the graph partitioning process in parallel.

The results of this evaluation are presented in Table 5.1 and in Figure 5.1. As expected, the default version of Signal/Collect with the default partitioning performs worst. Regarding the other three partitioning methods, however, the resulting execution times are too skewed to draw definite conclusions.

Besides this performance evaluation, the listed Signal/Collect version with the Push2 partitioner and the parallel scheduler partitioning every 2 seconds has been used to verify the correctness of the system by executing the algorithm and afterwards checking the result manually.

[3] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `6650edc04e6dbb88245dcd1b98d9b78a81b17ecc` for the `signal-collect` project and `f8246e99cbc095e1754834076f5e17675444eec6` for the `mte` project. The evaluation is contained in the file `VertexColoringEvaluation.scala` in the `mte` project on the accompanying CD-ROM.

[4] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `3cd7578ec40dd844ff603ae3c6dd565b950b2886` for the `signal-collect` project and `98613c2e895fbd0f468ba743c54a7a99c78181bc` for the `mte` project. The evaluation is contained in the file `MetisEvaluation.scala` in the `mte` project on the accompanying CD-ROM.

[5] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `fa2172d31391b5a2db28c479c5863bc9f3fcd206` for the `signal-collect` project and `e2fddaf6dfeb4379b45c6e89f970e2fe9587bd0f` for the `mte` project. The evaluation is contained in the file `VertexColoringPartitioningEvaluation.scala` in the `mte` project on the accompanying CD-ROM.

[6] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `fa2172d31391b5a2db28c479c5863bc9f3fcd206` for the `signal-collect` project and `12bacce8918177da81bbe452a2caedddf864157a` for the `mte` project. The evaluation is contained in the file `VertexColoringPartitioningEvaluation.scala` in the `mte` project on the accompanying CD-ROM.

Partitioning	Mean	SD	Median	MAD	Min	Max
Default	338163	465644	153790	145845	1541	1871059
METIS	20165	21650	10885	5862	1655	97351
Parallel	29002	64900	4697	955	2362	413022
Serial	23143	34851	5354	1690	1636	198717

Table 5.1: Mean, standard deviation (SD), median, median absolute deviation (MAD), minimum, and maximum execution times in milliseconds of 50 vertex coloring algorithm executions on a 10×10 grid graph with 3 colors using 3 compute nodes each with 1 worker

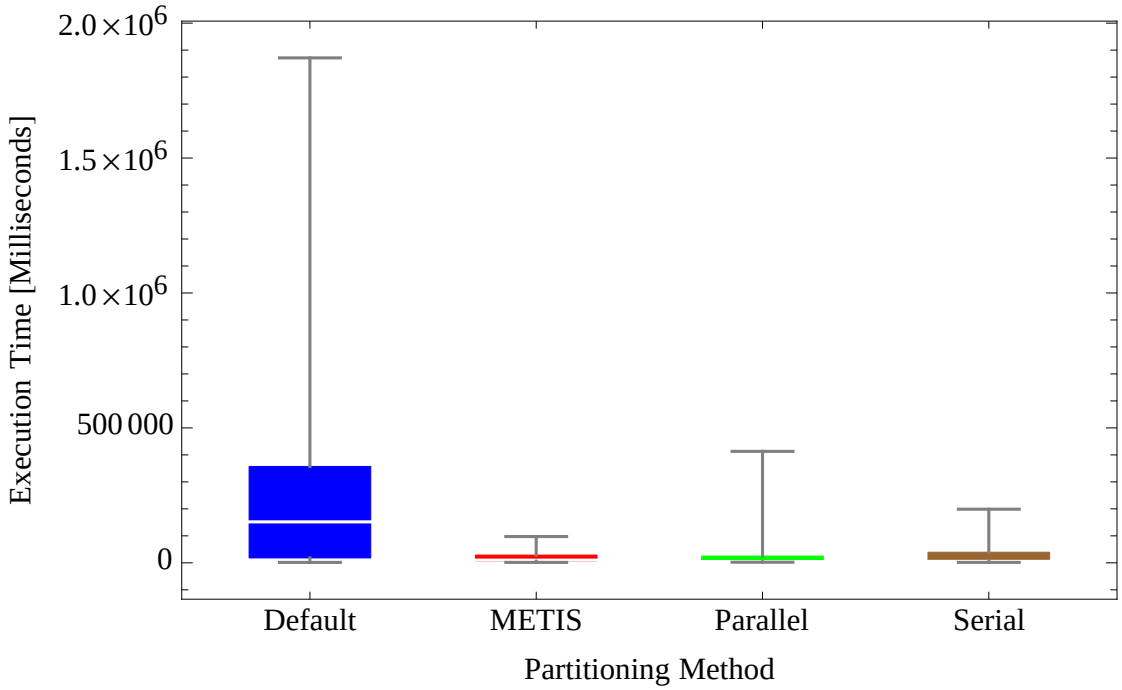

Figure 5.1: Execution times in milliseconds of 50 vertex coloring algorithm executions on a 10×10 grid graph with 3 colors using 3 compute nodes each with 1 worker. Note that the minimum execution time of all executions is 1541 ms.

5.3 TripleRush

TripleRush[7] is a store for Resource Description Framework (RDF)[8] triples built on top of Signal/Collect. As a triple store, TripleRush is a very interesting evaluation case for dynamic graph partitioning, because this system is long running and as such offers much potential for dynamic graph partitioning. It has been evaluated without

[7] *https://github.com/uzh/triplerush*
[8] *http://www.w3.org/RDF*

	Number of Compute Nodes	
Query	1	4
1	387	27320
2	2200	2463
3	258	402
4	4	9
5	3	227
6	7	34
7	206	10716

Table 5.2: Mean execution times in milliseconds when executing each of the 7 queries 10 times on the Lubm160 data set using different numbers of compute nodes each with 24 workers

graph partitioning using the Lubm160 benchmark [Zeng et al., 2013]. This benchmark consists of a synthetic data set of 21347999 triples and 7 queries. Because the data set, the queries, and the results using the default version of Signal/Collect are already available, it is appropriate to evaluate the partitioning functionality using this evaluation case. In chapter 4, it has been shown that the Push2 partitioner achieves a reduction of the number of remote messages of about 10% using a subset of the the benchmark's full data set. In this section, the results of two further evaluations using the Lubm160 benchmark are presented.

The 21347999 RDF triples of the data set result in a total of 47476987 vertices. The difference between the number of triples and the number of vertices is caused by the index structures created by TripleRush. In a first step,[9] each of the 7 queries of the benchmark is executed 10 times in an execution setting with 1 compute node and another one with 4 compute nodes. To initialize the executing virtual machines, the query is executed 10 times before the measured executions. The purpose of this experiment is to identify queries that take a significantly longer time when the data set is distributed over several compute nodes.

Based on the data in Table 5.2, it can be concluded that the execution times of query 1 and query 7 are significantly greater in the setting with 4 compute nodes than they are in the setting with only 1 compute node. In absolute numbers, query 1 offers the largest possibility to be improved by partitioning. Therefore, query 1 is a suitable candidate for the following evaluations.

[9] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `8c0d35a9dd6c4043c868648de7bc6240e8d8de37` for the `signal-collect` project and `bbfa3cfd0217a232364180cdc39dce0811923bb5` for the `triplerush` project. The evaluation is contained in the file `Lubm160SingleQueryEvaluation.scala` in the `triplerush` project on the accompanying CD-ROM.

Partitionings	Mean Execution Time [Milliseconds]
0	26522
1	26221
2	26215
3	26850
4	25741
5	27040
6	25197
7	26379
8	26722
9	26253
10	26603
20	24042
30	26374
40	26581
50	25624

Table 5.3: Mean execution times in milliseconds of 10 executions of query 1 on the Lubm160 data set when using 4 compute nodes each with 24 workers and partitioning the graph after each set of executions

5.3.1 Manual Partitioning

In the first evaluation,[10] the selected query is first executed 10 times. Afterwards, the graph is partitioned and the query is again executed 10 times. The last two steps are repeated 50 times. To initialize the executing virtual machines, the query is executed 50 times before the actual evaluation. Signal/Collect runs on 4 compute nodes each with 24 workers. Because of the repeated graph partitionings, it is to be expected that the runtime of the query decreases continuously.

The results of this evaluation are shown in Table 5.3 and graphically depicted in Figure 5.2. Almost all execution times are within 25–27 seconds. The results do not meet the expectations, because the execution times do not decrease significantly, but remain within the same range even after 50 partitionings.

[10] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `2b5cd6c9dfa16a75bdae219936693628e8a4583b` for the `signal-collect` project and `a4b2cd8ff33f884e011fac22f4a73241319a954f` for the `triplerush` project. The evaluation is contained in the file `Lubm160SingleQueryManualPartitioningEvaluation.scala` in the `triplerush` project on the accompanying CD-ROM.

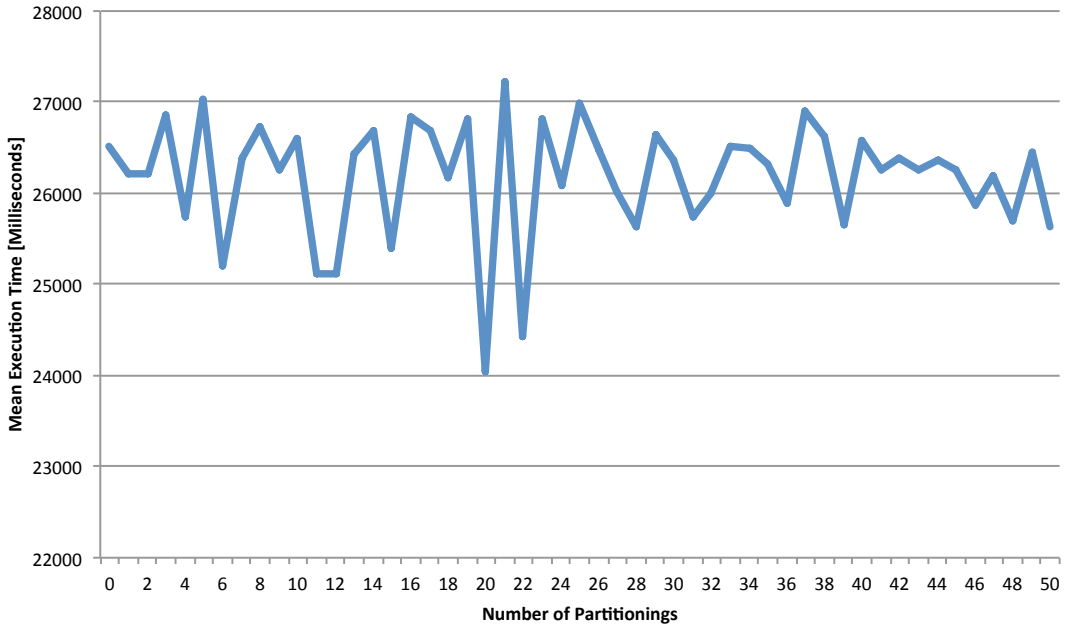

Figure 5.2: Mean execution times in milliseconds of 10 executions of query 1 on the Lubm160 data set when using 4 compute nodes each with 24 workers and partitioning the graph after each set of executions

5.3.2 Parallel Partitioning

In the second evaluation,[11] the parallel dynamic graph partitioning functionality is used. Query 1 of the Lubm160 benchmark is executed continuously over a period of one hour, while the graph is being partitioned in parallel every 10 seconds. To initialize the executing virtual machines, the query is executed 10 times before the actual evaluation. Signal/Collect runs on 4 compute nodes each with 24 workers. As before, we expect a continuously decreasing runtime of the query executions.

The results of this evaluation can be seen in Table 5.4 and are plotted in Figure 5.3. The execution times are not decreasing continuously. In comparison with the previous evaluation, the distribution of the execution times is higher, which could be explained by the fact that the query is executed only once. Furthermore, it could be the effect of the dynamic graph partitioning which runs in parallel.

[11] To reproduce the results of this evaluation, the corresponding Git revision identifiers are `b03084cb1a8cb909411842955784c6f6b297fb0a` for the `signal-collect` project and `8870714d9da4b2efec0412ed48d29d4c0ce3eade` for the `triplerush` project. The evaluation is contained in the file `Lubm160SingleQueryParallelPartitioningEvaluation.scala` in the `triplerush` project on the accompanying source code CD-ROM.

Number of Execution	Execution Time [Milliseconds]
1	31344
2	32508
3	29163
4	30254
5	29731
6	30661
7	28292
8	24181
9	23280
10	30068
20	26841
30	15746
40	28239
50	27667
60	19531
70	21105

Table 5.4: Execution times of query 1 on the Lubm160 data set in milliseconds using 4 compute nodes each with 24 workers and partitioning in parallel every 10 seconds

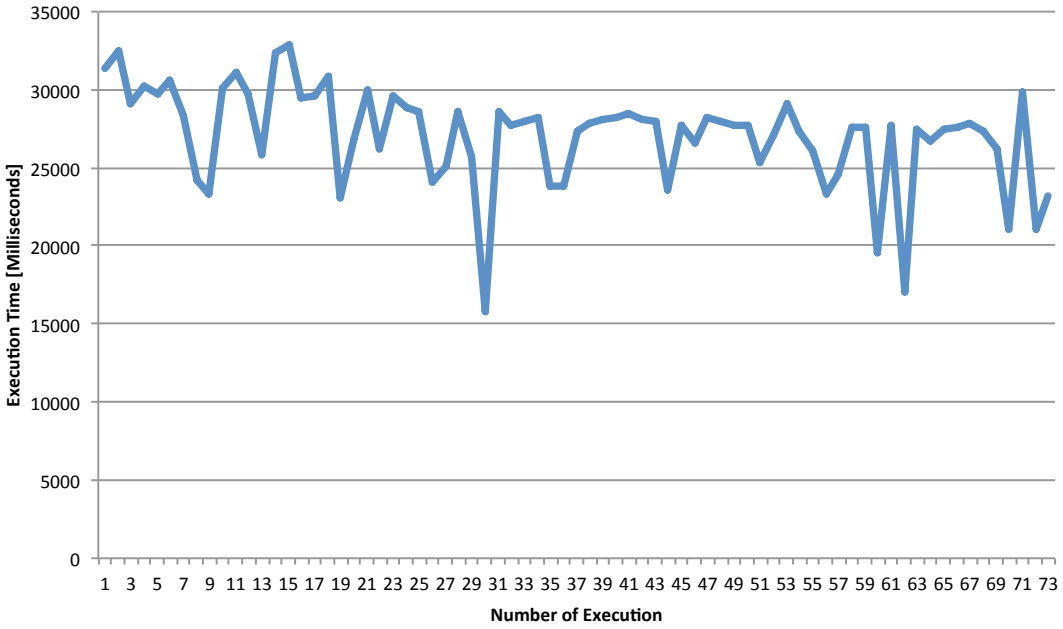

Figure 5.3: Execution times of query 1 on the Lubm160 data set in milliseconds using 4 compute nodes each with 24 workers and partitioning in parallel every 10 seconds

6

Discussion

After having presented the evaluation results in the previous chapter, the purpose of this chapter is to discuss them and offer answers to arising questions.

6.1 Vertex Coloring

From the results it can be concluded that the usage of the partitioning methods METIS and Push2 with serial and parallel schedulers improves the runtime performance of Signal/Collect. However, a quantitative statement is not possible due to the highly skewed data. For the same reason, it cannot be concluded that the Push2 partitioner performs better than METIS, although the median execution times would indicate this. The performed evaluation is not sufficient to make a clear statement about the performance gains to be achieved by the usage of the Push2 partitioner and how it performs compared to the graph partitioning tool METIS.

6.2 TripleRush

With regard to the TripleRush evaluations, no improvement of Signal/Collect's runtime performance can be observed. There are two general problems with Push2 that could cause the performance of the Push2 partitioner in the TripleRush evaluations.

The first situation is illustrated by Figure 6.1. There are two nodes N_0 and N_1 each with one of the two workers W_0 and W_1. W_0 is responsible for the two vertices 1 and 2, whereas vertex 3 is placed on worker W_1. It is also visible that 100 signals were sent from vertex 1 to vertex 2 and 50 signals from vertex 2 to vertex 3. Given this situation, Push2 attempts to move vertex 2 to worker W_1, because vertex 2 sent more messages to node N_1 than it did to node N_0. With regard to the number of remote signals, the resulting situation is worse because 100 instead of only 50 remote signals will be sent from worker W_0 to worker W_1. It is possible that vertex 1 will later be moved to worker W_1, too, but load balancing restrictions might prevent this displacement. Push2 is not able to recognize this kind of situation, because it only considers the number of sent messages from a vertex to workers and compute nodes.

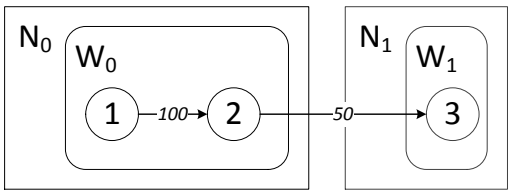

Figure 6.1: Situation in which Push2 takes a wrong decision

The second problem can be explained by the situation illustrated in Figure 6.2. There are three nodes N_{0-2} each with one of the workers W_{0-2}. Vertex 1 sent 50 remote signals to vertex 2 and vertex 2 sent 50 remote signals to vertex 3. In this situation, the Push2 partitioner might try to move vertex 1 from worker W_0 to worker W_1. However, the partitioner on node N_1 could at the same time move the vertex 2 to the worker W_2. This means that the reason for vertex 1 being moved is not existing anymore when the vertex arrives on its target worker, because it has also been moved.

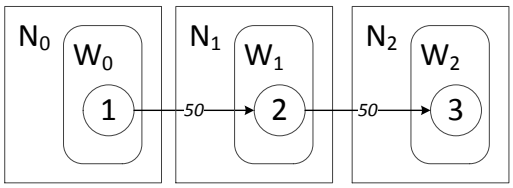

Figure 6.2: Problematic situation for Push2 when being executed on multiple compute nodes in parallel

Both these situations might occur when using the Push2 partitioner. They are possible explanations for Push2's performance in case of the TripleRush evaluations and might serve as hypotheses for further investigations.

6.3 Memory Consumption

In the evaluation cases, the memory consumption has not been explicitely measured. However, the amount of memory necessary for the data structures storing the number of signals can become problematic, when the number of vertices and workers is very large. To store the data used by the Push2 partitioner, a table mapping from vertex identifiers to an array containing the number of sent signals for each worker has to be maintained on each worker. For example, given 4 compute nodes with 24 workers, a vertex identifier size of 8 bytes, and 1 million vertices per worker this results in 147.5 Gigabytes of data on each compute node. In the TripleRush evaluation case, about 500000 vertices are stored on each worker.

Therefore, it can be concluded that the memory consumption will likely become problematic with larger graphs.

7

Future Work

Based on the experiences and evaluation results collected in this thesis, the following directions for future work can be pointed out.

Further evaluations of Push2 In this thesis, the Push2 partitioner has been evaluated using two evaluation cases. This is not sufficient to allow a definite verdict about the partitioner's performance. In a next step, the partitioner should be evaluated using other algorithms, different types of graphs, and different execution configurations, e.g. varying numbers of workers and other execution modes.

Partitioning algorithms In chapter 4, several partitioning algorithms have been presented and the best performing one has been discussed and evaluated in depth. The underlying framework, however, enables the development of a wide variety of algorithms due to its modularity. Collecting additional statistics about the vertices, workers, and nodes is simple, which further increases the possibilities for future algorithms. Therefore, other partitioning algorithms could be developed in the existing framework.

Execution in the cloud In this thesis, the dynamic graph partitioning functionality has been evaluated in a cluster system consisting of adjacent, homogeneous compute nodes. The execution and evaluation of Signal/Collect in a cloud environment is likely to increase the importance and impact of the partitioning functionality, because the cost of communication between compute nodes in such environments is higher.

Static graph partitioning The evaluation results for graphs partitioned by METIS demonstrate that static graph partitioning methods should definitively be investigated. In this thesis, the exception table approach has been described as solution for the arbitrary assignment of vertices to workers. Based on this approach, it is straightforward to integrate static graph partitioning approaches into Signal/-Collect. Promising ideas are domain-based partitioning, which assigns vertices to the same workers based on domain knowledge, streaming graph partitioning algorithms, which have been discussed in chapter 2, and the application of graph partitioning tools, e.g. METIS.

Comprehensive evaluation Both the dynamic and static partitioning functionalities should be systematically evaluated. As a result of such a comprehensive evaluation, there should be guidelines on how to use the partitioning functionality in Signal/Collect with regard to particular problems.

Automatic partitioning Based on the guidelines acquired in a comprehensive evaluation, Signal/Collect could be enhanced by automatic graph partitioning. This means that the system decides which partitioning should be applied in a given situation. Such an approach has already been implemented for static graph partitioning in [Khayyat et al., 2012].

8

Conclusions

In this thesis, Signal/Collect has been extended with dynamic graph partitioning to improve its runtime performance in distributed environments. In a first step, the adaptations to the existing framework have been described. Afterwards, four different partitioning algorithms have been presented and evaluated. The best performing approach, Push2, pushes vertices to the compute nodes to which they sent most messages to. It implements load balancing by restricting the range for the allowed number of vertices per worker dependent on the average number of vertices per worker in the system.

Push2 has been evaluated using two different evaluation cases, vertex coloring and the RDF triple store TripleRush. The evaluation of the approach led to two main conclusions. First, the approach improves the runtime performance of Signal/Collect in case of the vertex coloring evaluation case. However, not enough results have been collected to express the increase quantitatively. Second, the approach does not significantly improve the performance of query execution in the TripleRush system. Two general shortcomings of the Push2 partitioner have been pointed out and might serve as hypotheses for further investigations.

Finally, directions for future work have been pointed out. Besides further evaluations of Push2 and the development of other partitioning algorithms, static graph partitioning should be investigated. Executions in the cloud might increase the importance and impact of graph partitioning. The presented steps advance Signal/Collect towards a system that automatically selects the most sensible partitioning strategy for a particular problem.

References

[Averbuch and Neumann, 2013] Averbuch, A. and Neumann, M. (2013). Partitioning graph databases: A quantitative evaluation. Master's thesis, KTH Royal Institute of Technology.

[Bichot and Siarry, 2011] Bichot, C.-E. and Siarry, P., editors (2011). *Graph Partitioning.* ISTE Ltd and John Wiley & Sons, Inc.

[Bu et al., 2010] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296.

[Bykov et al., 2011] Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. (2011). Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14.

[Cai et al., 2012] Cai, Z., Logothetis, D., and Siganos, G. (2012). Facilitating real-time graph mining. In *Proceedings of the 4th International Workshop on Cloud Data Management*, pages 1–8.

[Chen et al., 2012] Chen, R., Yang, M., Weng, X., Choi, B., He, B., and Li, X. (2012). Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, pages 1–13.

[Choi et al., 2012] Choi, H., Um, J., Yoon, H., Lee, M., Choi, Y., Lee, W., Song, S., and Jung, H. (2012). A partitioning technique for improving the performance of pagerank on hadoop. In *7th International Conference on Computing and Convergence Technology*, pages 458–461.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[deLorimier, 2013] deLorimier, M. J. (2013). *GRAph Parallel Actor Language: A Programming Language for Parallel Graph Algorithms.* PhD thesis, California Institute of Technology.

[Devine et al., 2006] Devine, K. D., Boman, E. G., and Karypis, G. (2006). Partitioning and load balancing for emerging parallel applications and architectures. In Heroux, M. A., Raghavan, P., and Simon, H. D., editors, *Parallel Processing for Scientific Computing*, chapter 6, pages 99–126. Society for Industrial and Applied Mathematics.

[Ekanayake et al., 2010] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. (2010). Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818.

[Fard et al., 2012] Fard, A., Abdolrashidi, A., Ramaswamy, L., and Miller, J. A. (2012). Towards efficient query processing on massive time-evolving graphs. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 567–574.

[Fjällström, 1998] Fjällström, P.-O. (1998). Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, 3(10).

[Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3-5):75–174.

[Freitas, 2011] Freitas, F. d. (2011). Distributed signal/collect. Master's thesis, University of Zurich.

[Gehweiler and Meyerhenke, 2010] Gehweiler, J. and Meyerhenke, H. (2010). A distributed diffusive heuristic for clustering a virtual p2p supercomputer. In *2010 IEEE International Syposium on Parallel & Distributed Processing*, pages 1–8.

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

[Gregor and Lumsdaine, 2005] Gregor, D. and Lumsdaine, A. (2005). Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 423–437.

[Haller and Miller, 2011] Haller, P. and Miller, H. (2011). Parallelizing machine learning-functionally: A framework and abstractions for parallel graph processing. In *Second Annual Scala Workshop*.

[Ho et al., 2012] Ho, L.-Y., Wu, J.-J., and Liu, P. (2012). Distributed graph database for large-scale social computing. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing*, pages 455–462.

[Hoque and Gupta, 2013] Hoque, I. and Gupta, I. (2013). Lfgraph: Simpler is better for distributed graph analytics. Poster at the 10th USENIX Symposium on Networked Systems Design and Implementation.

[Kang et al., 2011] Kang, U., Tong, H., Sun, J., Lin, C.-Y., and Faloutsos, C. (2011). Gbase: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1091–1099.

[Kang et al., 2009] Kang, U., Tsourakakis, C. E., and Faloutsos, C. (2009). Pegasus: A peta-scale graph mining system - implementation and observations. In *9th IEEE International Conference on Data Mining*, pages 229–238.

[Karypis and Kumar, 1998] Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392.

[Khayyat et al., 2013] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. (2013). Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182.

[Khayyat et al., 2012] Khayyat, Z., Awara, K., Jamjoon, H., and Kalnis, P. (2012). Mizan: Optimizing graph mining in large parallel systems. Technical report, King Abdullah University of Science and Technology.

[Krepska et al., 2011] Krepska, E., Kielmann, T., Fokkink, W., and Bal, H. (2011). Hipg: Parallel processing of large-scale graphs. *ACM SIGOPS Operating Systems Review*, 45(2):3–13.

[Kyrola et al., 2012] Kyrola, A., Blelloch, G., and Guestrin, C. (2012). Graphchi: large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 31–46.

[Lin and Schatz, 2010] Lin, J. and Schatz, M. (2010). Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs*, pages 78–85.

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

[Low et al., 2010] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new framework for parallel machine learning. Published online at *http://arxiv.org/abs/1006.4990v1*.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146.

[Najork, 2009] Najork, M. (2009). The scalable hyperlink store. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, pages 89–98.

[Prabhakaran et al., 2012] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haridasan, M. (2012). Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*.

[Rahimian et al., 2013] Rahimian, F., Payberah, A. H., Girdzijauskas, S., Jelasity, M., and Haridi, S. (2013). Ja-be-ja: A distributed algorithm for balanced graph partitioning. Technical report, Swedish Institute of Computer Science.

[Ramaswamy et al., 2005] Ramaswamy, L., Gedik, B., and Liu, L. (2005). A distributed approach to node clustering in decentralized peer-to-peer networks. *IEEE Transactions on Parallel Distributed Systems*, 16(9):814–829.

[Redekopp et al., 2013] Redekopp, M., Simmhan, Y., and Prasanna, V. K. (2013). Optimizations and analysis of bsp graph processing models on public clouds. In *27th IEEE International Parallel & Distributed Processing Symposium*. To Appear.

[Salihoglu and Widom, 2012] Salihoglu, S. and Widom, J. (2012). Gps: A graph processing system. Technical report, Stanford University.

[Sarwat et al., 2012] Sarwat, M., Elnikety, S., He, Y., and Kliot, G. (2012). Horton: Online query execution engine for large distributed graphs. In *28th IEEE International Conference on Data Engineering*, pages 1289–1292.

[Shang and Yu, 2013] Shang, Z. and Yu, J. X. (2013). Catch the wind: Graph workload balancing on cloud. In *29th IEEE International Conference on Data Engineering*. To Appear.

[Shao et al., 2012] Shao, B., Wang, H., and Li, Y. (2012). The trinity graph engine. Technical report, Microsoft Research.

[Shun and Blelloch, 2013] Shun, J. and Blelloch, G. E. (2013). Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146.

[Stanton and Kliot, 2012] Stanton, I. and Kliot, G. (2012). Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: Graph algorithms for the (semantic) web. In *The Semantic Web - ISWC 2010*, pages 764–780.

[Stutz et al., 2013] Stutz, P., Strebel, D., and Bernstein, A. (2013). Signal/collect: Processing web-scale graphs in seconds. Under Review.

[Tsourakakis et al., 2012] Tsourakakis, C. E., Gkantsidis, C., Radunović, B., and Vojnović, M. (2012). Fennel: Streaming graph partitioning for massive scale graphs. Technical report, Microsoft Research.

[Wang et al., 2013] Wang, G., Xie, W., Demers, A., and Gehrke, J. (2013). Asynchronous large-scale graph processing made easy. In *Proceedings of the 2013 Biennial Conference on Innovative Data Systems Research*. To Appear.

[Xin et al., 2013] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In *Proceedings of the First International Workshop on Graph Data Management Experience and Systems*. To Appear.

[Yang et al., 2012] Yang, S., Yan, X., Zong, B., and Khan, A. (2012). Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*.

[Zeng et al., 2013] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale rdf data. In *Proceedings of the 39th International Conference on Very Large Data Bases*, pages 265–276.

[Zhang et al., 2011] Zhang, Y., Gao, Q., Gao, L., and Wang, C. (2011). Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14.

[Zhang et al., 2012] Zhang, Y., Gao, Q., Gao, L., and Wang, C. (2012). imapreduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68.

List of Figures

List of Tables