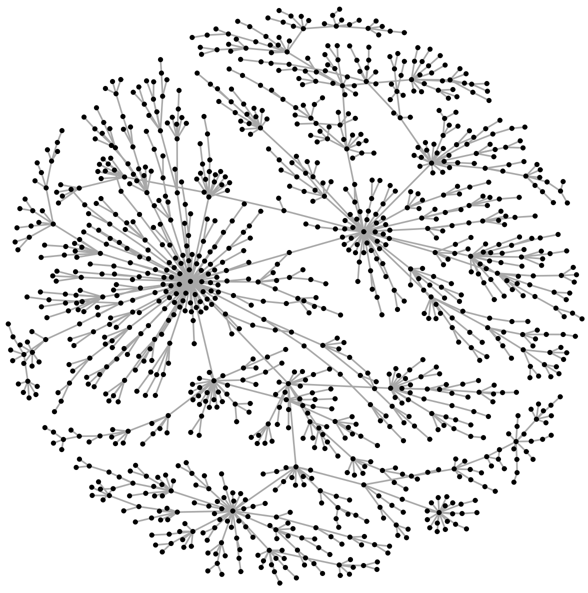
University of Zurich[UZH]

Large-Scale Social Network Analysis with the igraph Toolbox and Signal/Collect

Master Thesis September, 2015



András Heé
of Zurich, Switzerland

Student-ID: 08-982-142
andras.hee@uzh.ch

Advisor: **Daniel Spicar**

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

Acknowledgements

First and foremost, I would like to thank my advisor Daniel Spicar for his time, efforts and encouragements. His helpfulness and expert knowledge have been very supporting and motivating. Many thanks to Professor Abraham Bernstein for introducing me to the world of distributed systems and for giving me the opportunity to write the thesis at the Dynamic and Distributed Information Systems Group at the University of Zurich. I would like to express my thanks to Philip Stutz for developing Signal/Collect and for the technical support.

Furthermore, I am most grateful to my parents, Helen and Hansjörg Heé, whom both invested a lot of time and effort into me as an individual.

Zusammenfassung

Aufgrund der Entwicklung von skalierbaren und verteilten Systemen wurde in den letzten Jahren die Verarbeitung von grossen Netzwerken mit Millionen und Milliarden von Knoten und Verbindungen möglich. Oft fehlt diesen Platformen jedoch ein hoher Abstraktionslevel und eine Integration in bestehende Umgebungen. Beides sind Voraussetzungen für das effiziente Arbeiten von Datenanalysten. Unsere Arbeit besteht aus zwei Zielen: Einerseits haben wir auf der Basis des Signal/Collect[1] Frameworks eine generische Netzwerk Analyse Toolbox (NAT) entwickelt, welche die Integration zu dem verbreiteten Graphenanalysetool igraph[2] ermöglicht. Andererseits haben wir bekannte Algorithmen im Bereich der sozialen Netzwerkanalyse in das *vertex-centric* Programmierparadigma portiert.

[1]http://uzh.github.io/signal-collect/
[2]http://igraph.org

Abstract

In the last years, the processing of huge graphs with millions and billions of vertices and edges has become feasible due to highly scalable distributed frameworks. But, the current systems are suffering from having to provide a high level language abstraction to allow data scientists the expression of large scale data analysis tasks. Our contribution has two main goals: Firstly, we build a generic network analysis toolbox (NAT) on top of Signal/Collect[3], a vertex-centric graph processing framework, to support the integration into existing statistical and scientific programming environments. We deliver an interface to the popular network analysis tool igraph[4]. Secondly, we address the challenge to port social network analysis and graph exploration algorithms to the vertex-centric programming model to find implementations which do not operate on adjacency matrix representations of the graphs and do not rely on global state.

[3]http://uzh.github.io/signal-collect/
[4]http://igraph.org

Table of Contents

1 Introduction

Graphs, also known as networks, are fundamental data structures defined by a set of vertices where some pairs are connected by links. The study of networks has its origin in graph theory, but it has found applications in many disciplines. Graphs are used for modeling data in domains including computer science, protein networks, transportation networks, bibliographical networks or social networks [Elshawi et al., 2015]. The latter has traditionally been studied in sociology, where many network analysis methods originate from. The analysis of the role of vertices in networks and the properties of networks is known under the name social network analysis (SNA). In the last years, huge graphs with millions and billions of vertices and edges have become very common and with them designing scalable systems for their processing and analysis has become a challenging task for the big data research community.

Several frameworks address the problem of providing scalable graph processing platforms successfully by enabling parallelization and the distribution of the computation to multiple computers. But, the current systems are suffering in providing a high level language abstraction for expressing large scale data analysis tasks. Elshawi et al. [2015] compare the situation to the early days of the Hadoop[1] framework. The lack of declarative languages to allow data scientist the formulation of their queries efficiently has limited its practicality and the wide acceptance. Therefore, several systems like Pig[2] and Hive[3] have been introduced on top of Hadoop to fill the gap with higher abstractions.

In practice, end-users like data scientist need to execute computations which combine graph analysis with other analytics techniques. Concerning that the graph processing platforms are not able to connect with the ecosystem of other analytics systems, we address this problem by presenting a framework with an interface to the popular open source tool igraph[4]. Csardi and Nepusz [2006] introduce igraph as a network analysis library which can handle large graphs efficiently and which can be embedded into a higher level program or programming language providing a clean and comprehensive interface for manipulating, visualizing and analyzing graphs. Its core is implemented in

[1]https://hadoop.apache.org
[2]https://pig.apache.org
[3]https://hive.apache.org
[4]http://igraph.org

C/C++, but the tool provides wrapping libraries for the Python and the R programming languages. Both are widely-used for statistical and scientific programming.

Despite igraph using space and time efficient data structures and implementing current state-of-the-art algorithms, the package is not designed for distributed execution. This limits the size of networks that can be analyzed to the resources available to the local execution environment on which the algorithm is running.

Stutz et al. [2010] propose a vertex-centric programming model for synchronous and asynchronous graph algorithms designed for distributed parallel computation. Its implementation supports both parallelization through multiple processor cores as well as distribute computations to a server cluster.

This thesis aims to create an igraph API implementation that uses Signal/Collect graphs allowing to perform graph computations in a distributed system. For the prototype we limit the algorithms to a subset of the igraph functionality and focus on the implementation of algorithms used for social network analysis and graph exploration. The prototype integrates into the Python wrapper of the igraph package, but its architecture allows to extend it easily to support other languages. Calls to our custom igraph interface from Python are forwarded to the Signal/Collect graph representation transparently.

Our contribution has two main aspects. Firstly, we build a generic network analysis toolbox (NAT) on top of the Signal/Collect framework to support the integration into existing environments. Secondly, we address the challenge to port classic algorithms to the vertex-centric programming model to find implementations which do not operate on adjacency matrix representations of the graphs and do not rely on global state.

This thesis is structured as follows: The second chapter gives an overview on existing approaches and introduces the ideas behind the Signal/Collect framework. The third chapter explains the architecture of our generic framework. The fourth is the most comprehensive one discussing and evaluating all implemented algorithms and is organized into four sections. It begins with three *graph traversal* implementations, continuous with one *graph layout* algorithm, followed by two *graph transformations* and concludes with four *centrality measures*. Before concluding, the thesis discusses an outline of limitations and possible future work.

2

Related Work

In recent years researches have developed many techniques and frameworks to study very large graphs. Elshawi et al. [2015] give a solid overview to understand the challenges of developing scalable graph processing systems and introduce the state-of-the-art graph processing platforms. One of the most interesting approach is the vertex-centric one, which we discuss in the next section.

2.1 Thinking Like A Vertex

The publication of McCune et al. [2015] focuses on the very comprehensive discussion of the vertex-centric programming model. It is an established computational paradigm recently incorporated into frameworks to address challenges in the processing of large-scale graphs. Traditional implementations of big data tools, like the popular MapReduce implementation Hadoop, are not well-suited for iterative graph algorithms. In response, the vertex-centric approach tries to provide a natural interface allowing to program from the perspective of a vertex rather than the global graph. Such an interface improves locality, demonstrates linear scalability and provides a way to express and compute many iterative graph algorithms in an elegant way. The computation units generally only interact with data from adjacent vertices along incident edges. This allows for the design of an efficient decentralized architecture. Famous platforms are Pregel [Malewicz et al., 2010], Signal/Collect [Stutz et al., 2010], Giraph [Avery, 2011], GraphLab [Avery, 2011], GraphX [Xin et al., 2013] and Aster 6 [Simmen et al., 2014]. We won't discuss the systems in detail in this thesis but rather refer to the two survey papers mentioned above.

2.2 Existing Distributed Graph Analysis Libraries

There are a number of network analysis packages available. Popular examples are Graphviz[1], Gephi[2], igraph[3], networkX[4], SNAP[5] and Cytoscape[6]. Although having efficient implementations, all of them are not designed for distributed computation and therefore are limited by the resources of the local computer where the algorithm is running. Next, we present three frameworks which support scalable, distributed computing while providing implementations and high level abstractions for graph analysis tasks and therefore address the challenge to combine the two worlds of the scalable graph processing frameworks and the graph analysis libraries.

2.2.1 Aster 6

With Aster 6[7], Simmen et al. [2014] present a system which adds support for large-scale graph analytics to its repertoire of analytics capabilities. The solution also provides a vertex-centered interface to write graph analytics functions which can be executed using the multi-engine processing architecture with support for bulk synchronous parallel execution. The engine can be invoked from the context of a SQL query and thereby enables data scientist and business applications to express computations that combine large-scale graph analytics with techniques better suited to a different style of processing. Thanks to its support for standard SQL interfaces, Aster 6 is able to forward its results to external visualization tools like for example Tableau[8]. They include the Teradata SQL-MapReduce[9] framework allowing developers to write powerful SQL-MapReduce functions in languages such as Java, C#, Python, C++ and R.

2.2.2 GraphLab

With GraphLab[10], Low et al. [2014] present an open-source framework written in C++ using asynchronous execution mode. On top of GraphLab there is a graph analytics library[11] containing many standard algorithms. Dato Inc[12] was founded by Prof. Carlos Guestrin and supports the development of the project. The company provides a closed-source Python library on top of the C++ engine.

[1]http://www.graphviz.org
[2]http://gephi.github.io
[3]http://igraph.org
[4]https://networkx.github.io
[5]http://snap.stanford.edu
[6]http://www.cytoscape.org
[7]http://www.teradata.com
[8]http://www.tableau.com
[9]http://www.teradata.com/Teradata-Aster-SQL-MapReduce
[10]http://graphlab.org
[11]http://docs.graphlab.org/graph_analytics.html
[12]http://dato.com

2.2.3 Social Network Analysis with Signal/Collect

Keller [2014] has implemented social network analysis measures on the Signal/Collect framework. The tool is an extension of the existing graph tool Gephi[13] from where all the metrics can be executed. In contrast to his work our framework is a more generic approach to building a graph analysis service on top of Signal/Collect. The framework can be easily extended to support many client programming languages in addition to the provided Python wrapper. Furthermore, we investigate in finding vertex-centric algorithms for graph traversal and layouting problems and address some issues of Keller's centrality implementations.

2.3 Signal/Collect

Stutz et al. [2010] introduce Signal/Collect, again a vertex-centric programming model, where graph algorithms are decomposed into two operations. *Signaling* along edges informs neighbor vertices about changes and *collecting* the received signals updates the vertex's state. In contrast to many other implementations Signal/Collect supports both *synchronous* and *asynchronous* execution modes. The former is related to the Bulk Synchronous Parallel (BSP) paradigm. There is a global synchronization between the computation steps consisting of a signal phase and a collect phase. This ensures that the signal and the collect phase of different vertices never overlap.

In the asynchronous execution mode there are no guarantees about the order in which the operations on vertices get executed. Stutz et al. argue that depending on the algorithm, this may perform better, because it has the potential to propagate information across the graph faster and is less susceptible to oscillations in converging algorithms. We will investigate the performance differences for our algorithms in the fourth chapter.

Signal/Collect supports both parallelization using multiple cores as well as distribution over cluster of server nodes using the Akka[14] actor framework.

[13]http://gephi.github.io
[14]http://akka.io

3 Architecture

The framework consists of four separate modules: *Thrift Service Interface*, *Python Client*, *Signal/Collect Server* and *Evaluation*. Each is located in its own repository. Figure 3.1 gives an overview of all components and their relationships. In the following sections we will discuss them in detail.

3.1 Apache Thrift Service Interface

Apache Thrift[1] is a framework for building cross-language services. It provides a domain specific language (DSL) for defining service interfaces, a code generation engine and a runtime environment to communicate seamlessly between a huge set of supported languages. The Thrift definition file (`NATService`) is the interface of the service and specifies the contract of the API. In the DSL all supported methods and structs of the service are listed. The generated code for the individual languages integrates very well into the native language environment, allowing to develop cross-language services almost transparently. In the scope of this thesis we implemented one client for our NATService using Python. Due to the modular service definition and the huge language support of Thrift it could be extended easily to further languages.

The runtime library supports a number of protocols to establish the binary remote procedure call with a lower overhead than other alternatives like REST or SOAP as a performance comparison of Sumaray and Makki [2012] shows.

3.2 Python Client

The *Python Client* implements a graph library. The core is the `SCGraph` class which provides a subset of the interface of the igraph library. Providing the same method signatures, the user can use this class to build a graph and execute algorithms on it using the same methods as if she would use the igraph library. In contrast, the commands are executed transparently on a (possibly distributed) Signal/Collect server instance. It is possible to configure the SCGraph to create and maintain a local igraph instance of the

[1]https://thrift.apache.org

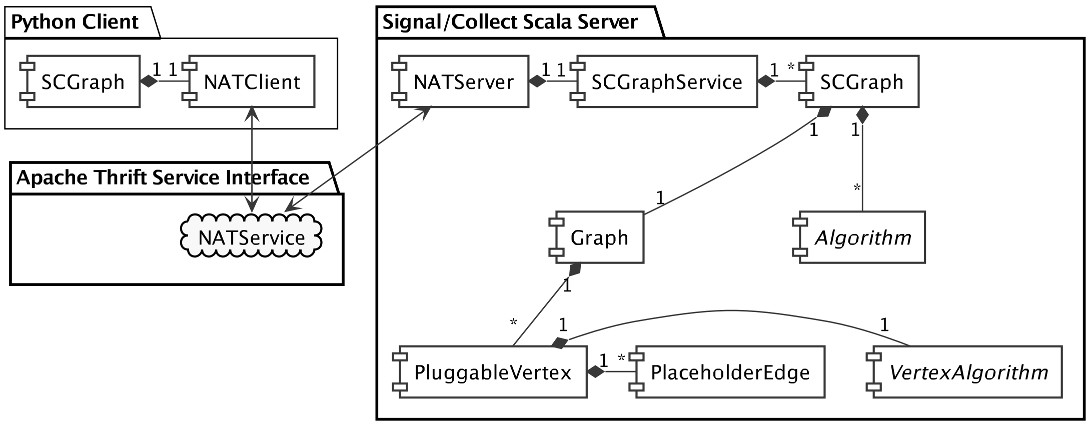


Figure 3.1: Components of NAT

same graph. This enables the user to choose which algorithms should be run locally and which distributed. Also, the full extend of functionalities igraph can be accessed.

`SCGraph` has an instance of the `NATClient` class which is responsible for sending and receiving the binary messages from the `NATServer` using the Thrift protocol.

We support multiple Python `SCGraph` instances in parallel. The framework takes care of the initialization of the Signal/Collect counter part on the server. Furthermore, there is no need to shutdown a Signal/Collect instance manually. The garbage collection on Signal/Collect is triggered automatically after Python deletes an instance.

The Python client contains acceptance tests to assess the correctness of the implementations.

3.3 Signal/Collect Scala Server

The Signal/Collect server is the core of the framework. `NATServer` launches a socket server to be able to answer requests from a `NATClient`. It has an instance of `SCGraph-Service` which implements the interface provided by the Thrift Service definition `NAT-Service`. It handles the creation and deletion of all `SCGraph` instances and delegates incoming service commands to the correct instance. Each service call has to include a unique identifier of the graph in order to be able to establish the synchronization between the Signal/Collect graphs and the graph representations on the client side.

`SCGraph` is the center of the framework from an algorithmic perspective. It implements the logic of the service commands. It has an instance of the proper Graph provided by the Signal/Collect framework and lazily instantiates the algorithms when required.

3.3.1 Programming Model

In Signal/Collect algorithms are designed from the perspective of vertices and edges. An algorithm is developed by extending one of the default implementations of a vertex and edge provided by the library. The instances of the concrete vertex and edge classes are then registered and distributed by the framework. One big challenge with this implementation by inheritance approach is the execution of different algorithms without the need of rebuilding and redistributing of the entire graph structure.

We use a similar approach as Strebel [2013] proposes. We implement a vertex `PluggableVertex` which implements the `Vertex` interface provided by Signal/Collect. Rather than implementing any logic itself, it delegates all calls to an implementation of the `VertexAlgorithm` which can be plugged in at run-time using the state pattern as originally described by Gamma et al. [1994]. The abstract VertexAlgorithm class also has an interface analog to the original Signal/Collect Vertex. This allows us to execute multiple algorithms while preserving the existing graph with all its vertices, edges and attributes. Changing the algorithm has a complexity of O(n) while n is the number of vertices, but can be executed in parallel.

The basic idea of the Signal/Collects programming model is that the computations are executed on a graph, similar to the actor model. The vertices are the computational units that interact by the means of signals which flow along the edges. The vertices collect the incoming signals, use them for computation and then signal the neighbors in the graph. In the default programming model of Signal/Collect a subclass of `Vertex` implements a custom `collect` method and a subclass of `Edge` may override a custom `signal` method. Shifting the signaling operation to the Edge is syntactical sugaring of the programming model. In the underlying implementation the outgoing edges are attached to the source vertex and distributed to the same worker. In our implementation both signal and collect operations are handled by the `PluggableVertex` which delegates them to the algorithmic specific `VertexAlgorithm`. We are using a `PlaceholderEdge` with does not contain any algorithmic logic and just forwards all signals. Besides allowing a more memory efficient implementation we can limit the update process for a new algorithm to the replacement of the `VertexAlgorithm`. Having a custom signaling function on the edges would require to iterate through every edge as well.

Signal/Collect provides two default implementations for vertices[2]. The `DataGraphVertex` is suitable for algorithms that iteratively update states associated with vertices and edges. Often, in iterative computations old values get replaced by newer ones . In the collect function of a DataGraphVertex only the most recently received signal for each incoming edge is delivered. The `DataFlowVertex` is designed for dataflow computations where data is routed through a network of processing vertices and for algorithms which rely on the delivery of a signal. In a dataflow computation no signal should ever be lost and the signals are ordered by their time of arrival. In our framework we adopt this differentiation and provide the same logic.

In addition, both `PluggableVertex` and `PlaceholderEdge` contain a Map to support the storage of attributes.

[2]https://github.com/uzh/signal-collect/wiki/Default-Vertex-Types

3.3.2 Execution Model

Figure 3.2 illustrates the life-cycle of a SCGraph. In a first step the Graph is created and its structure is built by adding `PluggableVertices` and `PlaceholderEdges`. We then can execute multiple `Algorithm` instances on it by setting the corresponding `VertexAlgorithm` on all vertices. Some algorithms need additional `HelperEdges` for the execution which are added before and removed after the Signal/Collect operation itself. Last the algorithm extracts or aggregates the relevant data from the states of the individual vertices.

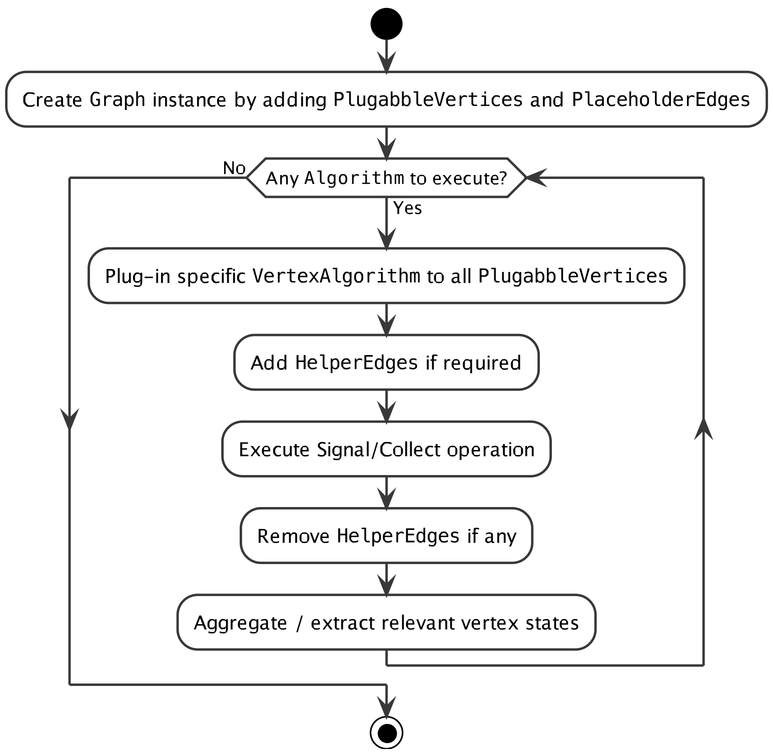

Create Graph instance by adding PlugabbleVertices and PlaceholderEdges
No
Any Algorithm to execute?
Yes
Plug-in specific VertexAlgorithm to all PlugabbleVertices
Add HelperEdges if required
Execute Signal/Collect operation
Remove HelperEdges if any
Aggregate / extract relevant vertex states

Figure 3.2: Algorithm Execution Flow

3.3.3 Testing

For every implemented algorithm there are many unit tests included to ensure the correct computation of the result. We have created small sample graphs to test the algorithms against following graph properties: directed and undirected, connected and disconnected, with and without attributes and edge weights and with and without cycles. In total the test suite contains 99 tests.

3.3.4 Data Loading

Himsolt [1997] proposes the Graph Modeling Language (GML), a file format for storing graphs. It is hierarchically structured and allows the description of vertices, edges and attributes on them. There already already a GML Parser[3] for the Signal/Collect framework. The implementation parses the entire file into memory of the master worker and builds a distributed Signal/Collect graph. We implemented an alternative which iteratively builds the distributed graph using functional pattern matching without loading the entire graph into the local memory to support graphs larger than the memory of the master worker node.

3.4 Evaluation Module

The framework includes an evaluation module which allows to assess the scalability and the degree of parallelization of the algorithms. It enables the deployment of an evaluation batch, testing numerous combinations of properties like different graphs in GML format, execution modes (synchronous or asynchronous) or server configurations (number of nodes and cores) within one click to the server. After the calculation, the results can be pulled from the server in form of a csv file.

[3]https://goo.gl/8GLv0z

4 Algorithms and Evaluation

In this chapter we discuss the algorithms we wrote for the framework described in the previous chapter. They are organized into four sections. We start with three *graph traversal* implementations, present one *graph layout* algorithm, continue to two *graph transformations* and conclude with four *centrality measures*. For every algorithm we shortly introduce the addressed problem with its mathematical definition, followed by our implementation in Signal/Collect. Finally, we evaluate the performance.

Synthetic Evaluation Graphs We have chosen to generate three different types of graphs to observe the behavior of the algorithms for different topologies. Generating synthetic graphs has the advantage that we can assess the scalability of the algorithms by scaling up the graph sizes and compare the heterogenous structures. We use the *Erdős Rényi* model to simulate random graphs proposed by Erdös and Rényi [1960], the *Barabási Albert* model for scale-free networks using a preferential attachment mechanism described by Albert and Barabási [2002] and the *Watts Strogatz* model to produce graphs with small-world properties formulated by Watts and Strogatz [1998].

Table 4.1 summarizes relevant metrics for the graphs to understand their differences. The graphs are generated in the way that they have about the same number of edges for a given number of vertices. This allows to compare the results of the algorithms between the graphs and leads to a homogenous *density*, the ratio of the number of edges and the number of possible edges. The Watt Strogatz model has a higher clustering coefficient than by random chance. Similar to the Erds Rényi model the degree distribution is a Poisson distribution leading to a relatively homogeneous network topology. The vertices have degrees in a similar range. In contrast, in the Barabási Albert model each new vertex is attached with a probability proportional to the degree of existing vertices. This results in a degree distribution following the power law. A few vertices have very large degrees, whereby the majority has small values. This phenomenon can be observed in many networks including the world wide web, citation networks and some social networks.

Evaluation Process We focus our evaluation on two properties. In the context of this paper, under the term *scalability* we refer to the behavior of the algorithms scaling up the graph sizes. *Parallelization* measures the proportional speedup by adding more

Metric	Erdős Rényi	Barabási Albert	Watt Strogatz
Density	0.008	0.008	0.008
Transitivity	0.008	0.026	0.473
Avg Path Length (v=1000)	5.921	2.626	5.131
Degree Distribution	Poisson	Power-Law	Poisson

Table 4.1: Synthetic Graphs Used For Evaluation

computational power. Akhter and Roberts [2006] and Appuswamy et al. [2013] give good introductions into parallel computing. We want to invest how the algorithms and the underlying Signal/Collect framework perform in terms of vertical and horizontal scaling. Vertical scaling (also known as scale-up) adds more processors and memory to one computation unit. Horizontal scaling adds more servers. The later is often more economical, can theoretically scale infinitely and is easier to run fault-tolerance. Vertical scaling however has the advantage that it can profit from very fast communication speed in contrast to the higher network latency between servers.

For the evaluation we use a cluster with a total of 16 machines provided by the University of Zurich. Job submission is controlled by the Slurm resource scheduling system[1]. Each machine has 128 GB RAM and two E5-2680 v2 at 2.80GHz processors, with 10 cores per processor. The machines are connected with 40Gbps Infiniband. We also use a commodity laptop (MacBook with two 2.5 GHz Intel Core i5 processors and 8 GB RAM) to evaluate the performance of the Python igraph implementation. It seems to be unfair to use different hardware for the evaluation, but should reflect the use case described in the introduction. The data scientist usually does not work directly on a scaled-up server, but on her local machine. Our framework should enable the usage of distributed parallel programming transparently. The measured runtime includes the execution of the algorithm and the aggregation of the results, but excludes parsing and distributing the graph.

For some algorithms we evaluate the performance of two execution modes Signal/Collects provides. From the four available we select the *synchronous* mode, with synchronized computation steps and the default *optimized asynchronous* mode, where there is one synchronous signal operation before switching to an asynchronous execution schedule.

4.1 Graph Traversals

Graph traversal is the problem of iterating over vertices in a graph in a predefined order. A lot of algorithms are building upon graph traversal. The applications are numerous and Russell and Norvig [1995] mention for example searching for the best path to a target vertex for all kind of routing applications, searching for the best alternative in decision and game theory setups like the minimax decision rule in a two-player game search,

[1]http://slurm.schedmd.com

determining whether a graph is a directed acyclic graph or enumerating all reachable vertices from a target for garbage collecting.

Generally, given a single vertex or a set of vertices, the goal is to find all reachable vertices from these roots. In an undirected graph we can follow all edges to find the target, in a directed one we are restrict to the outgoing ones.

Russell and Norvig [1995] discuss two crucial properties for graph traversal algorithms. If a strategy is *complete* it always finds a solution when there exists one. If it is *optimal* it finds the highest-quality solution when there are several different ones. We will apply these properties on the three algorithms we discuss in the next sections.

4.1.1 Breadth First Search (BFS)

The *breadth first search* is fundamental for the graph theory. It appears in various applications where shortest paths are searched.

Algorithm

According to Weisstein [2015a] the breadth first search explores all vertices adjacent to the current vertex before moving on. In other words, the algorithm starts at a given single root vertex and expands the neighbor vertices level by level. The order reflects the distance from the vertices to the root vertex. All the vertices at depth d in the search tree are expanded before the vertices at depth $d + 1$. The edge weights are not considered and treated equal to 1.

The algorithm is *complete* as it always finds a solution if there exists one. It is also *optimal* if we assume that deeper solutions are less optimal. To support varying edge weights we can introduce the *uniform cost search* (UCS), also known as *Dijkstra's algorithm*.

Rather than treating all edge weights homogeneously, the uniform cost search expands the vertex first having the smallest path cost represented by the sum of all edge weights. It is *complete* and *optimal* under the assumption that all path cost are strictly positive. Otherwise it can get stuck in infinite loops.

Implementation

Implementing BFS using the Signal/Collect programming model is very straightforward. The state of each vertex is represented by a path containing the current path length and the id of the parent vertex in the search tree. Storing the parent is not required for building the path sequence, but allows an efficient representation of the final result according to the igraph API which returns a tuple with the vertex ids in visited order, the start indices of the layers in the vertex list and the parent of every vertex in the BFS.

Figure 4.1 displays the process of the implementation. Before execution, the master sets the state of the root vertex to a path with length of 0. In parallel on each vertex, the signal function iteratively sends its current path length plus 1 to all its outgoing

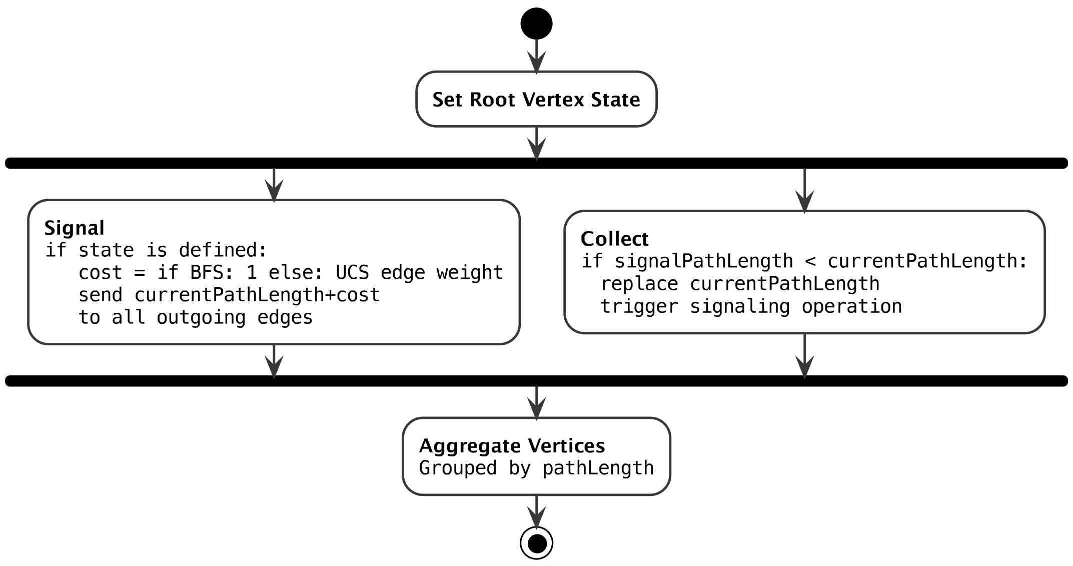

Set Root Vertex State
Signal
if state is defined:
cost = if BFS: 1 else: UCS edge weight
send currentPathLength+cost
to all outgoing edges
Collect
if signalPathLength < currentPathLength:
replace currentPathLength
trigger signaling operation
Aggregate Vertices
Grouped by pathLength

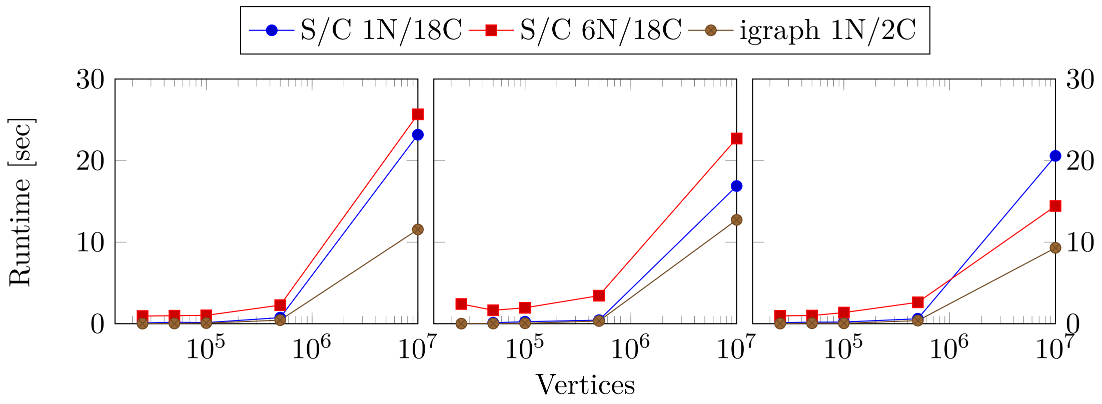
Figure 4.1: BFS Algorithm Activity Diagram

vertices. All incoming signals are collected and their length are compared to the current knowledge. If the path is shorter it will be stored and the signal function is triggered again. UCS can be implemented analogously. The only difference is that we sum up the edge weight rather than incrementing the path length by 1. The algorithm builds on top of the `DataFlowVertex` semantics, since it is crucial to receive every signal sent.

Evaluation

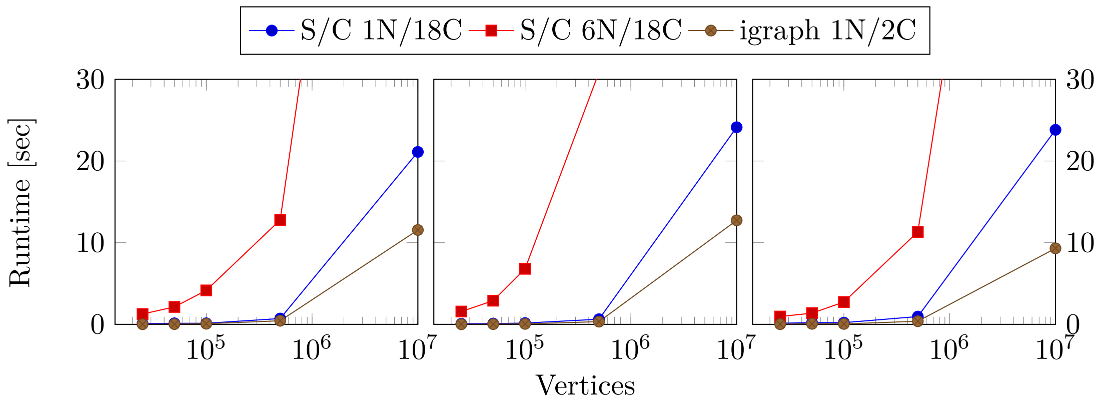
Figures 4.2 and 4.3 plot the runtimes of the BFS algorithm for different graph sizes. For this evaluation we use undirected versions of the discussed graphs to ensure that the root vertex can reach many other vertices. The former uses the Signal/Collect's *synchronous* execution mode and the second one the *optimized asynchronous* one. The x-axis, representing the number of vertices, is plotted in logarithmic scale. We observe that the asynchronous mode performs much worse in the scale-out setup. In this test scenario, the runtime for the 6 nodes setup is strictly larger than the one for the single node one. For the Barabási Albert graph with 500′000 vertices and 6 server nodes using the synchronous execution mode results in a speedup of more than 460% compared to the asynchronous one. In the case of the Erdős Rényi the largest graph size with 10 millions vertices couldn't even be computed in the asynchronous mode due to a buffer overflow of the Akka framework. In the single node setup the difference is smaller and for the Barabási Albert graph the asynchronous mode is even slightly faster. Our interpretation for this observation is that the synchronous mode allows to send the message in larger bulks over the network which turned out to be more important than the theoretical benefit of the asynchronous mode.

The igraph implementation performs very well on our commodity laptop. The runtime



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.2: BFS Scalability Evaluation (Synchronous Execution Mode)



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.3: BFS Scalability Evaluation (Optimized Asynchronous Execution Mode)

is constant for all three graph types and outperforms both the 1 node and the 6 nodes Signal/Collect implementations. However, loading the 10 million GML file on the laptop is at the upper limit of the memory capabilities. If the use case requires to scale up to larger graphs the usage of the Signal/Collect framework would pay off since the memory is not the limiting factor anymore and the runtimes are still in a feasible range.

Figure 4.4 illustrates the degree of parallelization running the algorithm in five different server configurations for a fixed graph size. The three bars on the left display the vertical scaling possibilities using 1, 9 and 18 cores. The three on the right illustrate the horizontal scaling using 1, 3 and 6 server nodes witch each 18 cores. The patterns for the three different graph types are very homogenous. It is striking that scaling-up the number of cores on a single machine brings a huge benefit to the runtime. For the Barabási Albert graph using 9 cores instead of 1 is more than 87% faster. The speedup decreased from 9 cores to 18. What is very surprising is that scaling the system out

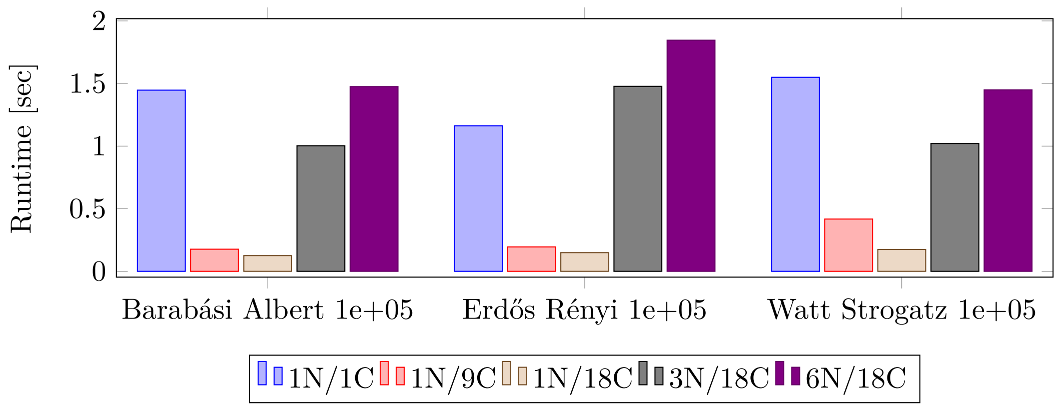


Figure 4.4: BFS Parallelization Evaluation (Synchronous Execution Mode)

by adding 3 or even 6 other nodes leads to a higher runtime than using only one node. Although the severs are connected by a fast 40Gbps Infiniband, the effect of the network costs and latency overhead is larger than the additional computation power. The only scenario where the 6 server nodes can outperform the single node configuration is the Watt Strogatz graph with 10 millions vertices in the synchronous execution mode.

We conclude that the runtime and the best configuration is difficult to estimate. Scaling-up on a single machine is beneficial. The optimized asynchronous execution mode is very fragile running on a scaled-out environment and vertical scaling does not result in better performance per se.

Future Work

We assume that a message combiner on worker level could improve the horizontal scaling drastically. We could group the signals by the target vertex id and deliver only the one with the smallest score. All other signals would be discarded in the collect method of the target vertex and can therefore be dropped already before sending them across the network. Having the possibility to run multiple algorithms consecutively is a requirement and should be supported by our framework. We have not implemented an algorithmic specific message bus due to the fact that we could not find a solution to plug it into an existing graph.

4.1.2 Depth First Search (DFS)

The *depth first search* is another graph traversal mechanism often used in the artificial intelligence domain. It can for example be used to build backtracking algorithms for combinatorial and constraint satisfaction problems.

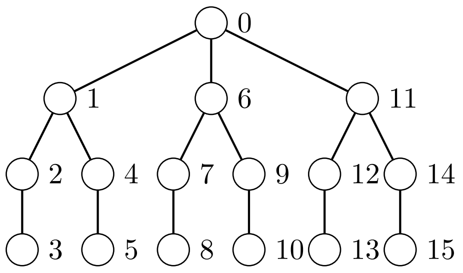


Figure 4.5: Enumerated Vertex Ranking For All Possible DFS Paths ($n = 4$)

Algorithm

Sequential Traversal Weisstein [2015b] defines DFS as a search algorithm that explores the first child of a vertex before visiting its siblings. The naive idea to implement a DFS without a global adjacency matrix using Signal/Collect is to iteratively visit the first child up to a leaf vertex and then backtrack in order to visit the siblings in the correct DFS order. Unfortunately, this idea results in a sequential implementation without using any benefits of the distributed framework.

The idea is to develop an algorithm which can assign a rank to each vertex and can be executed in parallel. Ordering the vertices according to the ranks should sort them according to the DFS order. The ranks are not required to be continuously increasing, only the order is of relevance.

Enumerating All Structural Combinations Given a graph with n vertices we can build a search space tree sp enumerating all possible search paths which could occur for a general graph considering all structural combinations. Figure 4.5 shows an example for a graph with 4 vertices. Its height is $n-1$. Starting from the search root, in a general case the vertex could be connected to all remaining vertices. This leads to a branching factor of $n-1$ at a depth of 0. At a depth of 1 the branching level is reduced by one since at least two vertices are already expanded. Generally speaking a vertex in the tree has a branching factor bf which depends on the depth d:

$$bf_{sp}(v) = n - d - 1 \tag{4.1}$$

We can calculate the number of vertices n_{sp} in the tree:

$$n_{sp} = \lfloor e * (n-1)! \rfloor \tag{4.2}$$

Having this enumeration of all cases for a general graph we can design a function to assign a unique rank to each vertex with the restriction that the ordering follows the DFS sequence. Given n, the number of vertices in the graph, the depth $d(v)$, the child position $k(v)$ and rank of its parent $pr(v)$ the exact rank for a vertex $r(v)$ is:

$$r(v) = pr(v) + 1 + \lfloor e * (n - d(v) - 1)! \rfloor * k(v) \tag{4.3}$$

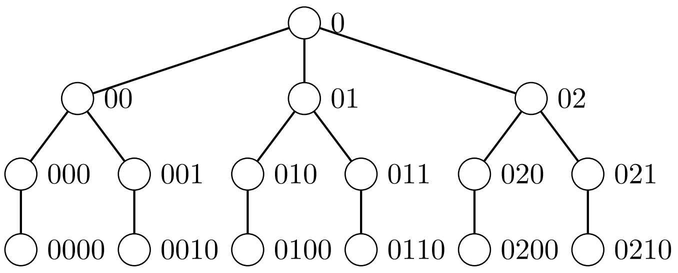


Figure 4.6: Child Sequence Vertex Ranking For All Possible DFS Paths ($n = 4$)

With this equation we can assign to every possible case a unique natural number in parallel without being limited to a sequential backtracking bottleneck. The problem is that the representation of one rank value for one vertex has a space complexity of $O(n!)$, which is not feasible at all. With $32bit$ we could represent a rank for a graph with only 12 vertices. With $64bit$ it could be increased to 20 vertices. The total space complexity for all vertices would be $O(n * n!)$. Another downside of the approach is the need to calculate the factorial of n which has a time complexity of $O(n!)$ by definition. However, this can be improved using the sterling approximation.

Child Indexes Queuing Enqueuing the child index is another approach to get a unique identifier which, when sorted, represents the DFS order. Starting again from the search root vertex, in every level the current child position is enqueued to the parent rank. Having the parent rank vector $pr(v)$ and the child position $k(v)$ of a vertex v we can compute its rank vector $r(v)$ as follows:

$$r(v) = [pr(v), r(v)] \tag{4.4}$$

The size of the rank vector is at maximum n. An element in the vector has to represent the range from minimum to the maximum branching factor in the tree, which is $n-1$ in the worst case. This leads to a worst case space complexity of $O(n^2)$. On average this is lower since the branching factor is much smaller for most vertices in most graph types. The time complexity for the rank calculation is $O(1)$ using a correct data structure, which is also a huge improvement to the factorial approach discussed before.

Implementation

At this point, we limit the detailed discussion and evaluation to the child indexes algorithm since it is the most promising approach for the Signal/Collect framework out of the three proposed ones. The naive sequential implementation does not fit well for the parallel computing framework Signal/Collect and the absolute rank computation with the faculty space footprint is not feasible in practice.

Child Indexes Queuing Figure 4.7 shows the flow of the parallel rank propagation. If the vertex has a rank it sends it to all its outgoing edges. When a vertex receives

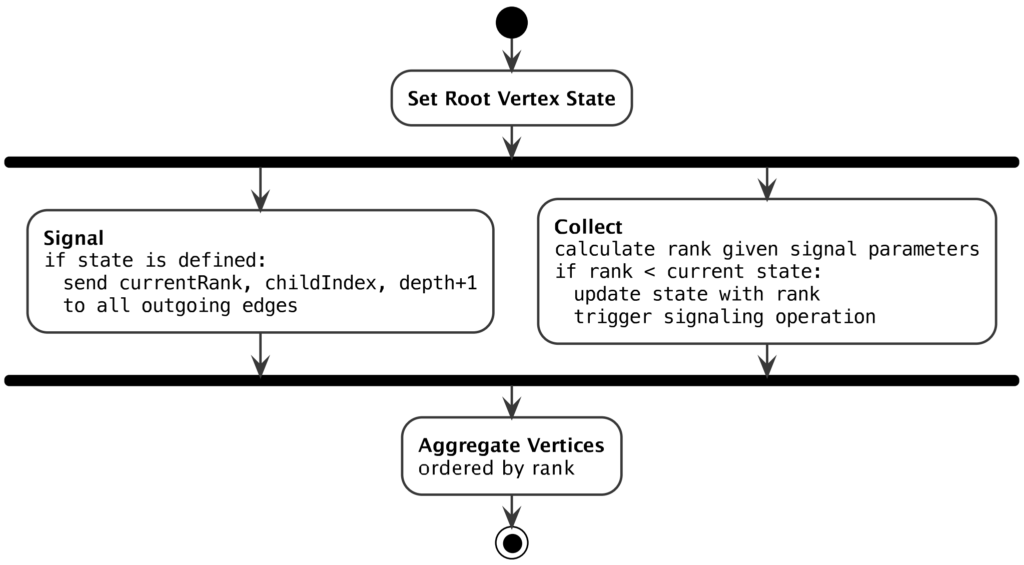


Figure 4.7: Child Indexes Queuing DFS Algorithm (Activity Diagram)

a signal it calculates the rank from the given signal parameters and compares it to its current rank. If it hasn't yet a rank associated or the current rank is larger than the received one it will update it and trigger the signaling operation again. Eventually, the ranks converge and no messages are sent any more. It is implemented as a data flow to ensure every signal is collected.

Evaluation

In the Python igraph there is no DFS implementation, therefore igraph runtimes are missing in our comparison.

Child Indexes Queuing Analogous to the evaluation of the BFS, Figure 4.8 visualizes the scalability using the three synthetic graphs. However, the x-axis is not in logarithmic scale. The runtime for the scaled-out version explodes with increasing graph size. This is reasonable since not only the number of the signals increases with the graph size, but also the size of one signal itself containing the encoded child index sequence. Sending large signals across the network has a larger impact than managing it inside one computer. The asynchronous mode did perform much worse, so we omit the discussion of it for this algorithm.

We are surprised by the bad performance of this algorithm. For a small-world Watt Strogatz graph with only 7500 vertices the algorithm needing about 280 seconds to capture the correct DFS sequence. Assuming that there are no major flaws in the implementation, we suspect that the biggest issue with the child index propagation algorithm is that the messages are sent in a BFS order. Figure 4.9 illustrates how the DFS ranks converge. We split the process into three steps using a directed graph.

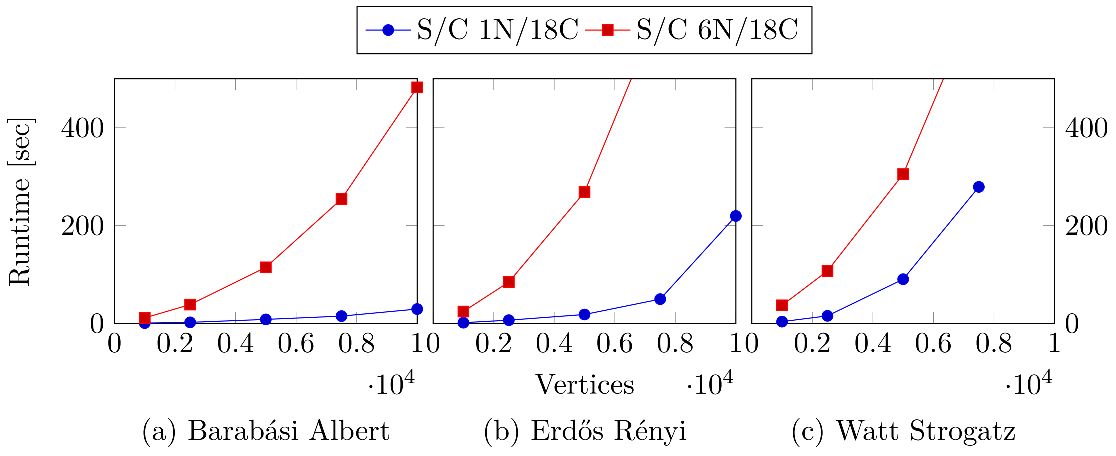


Figure 4.8: Child Indexes Queuing DFS Scalability Evaluation (Synchronous Execution Mode)

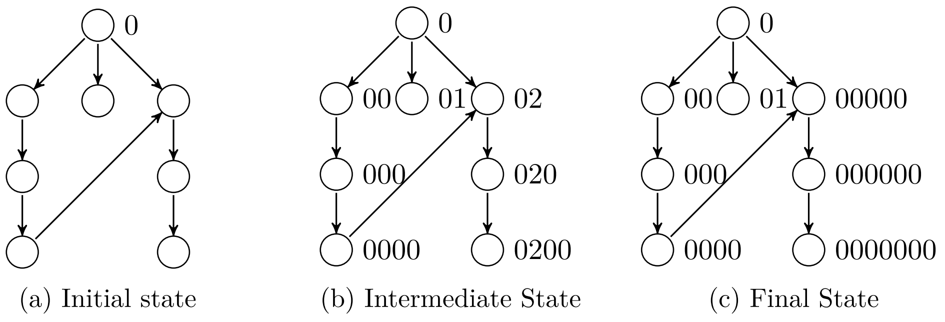


Figure 4.9: Computation Flow of Child Indexes Queuing Algorithm

Initially, the state of root vertex is set to 0. After a few Signal/Collect steps we obtain the intermediate state. Every vertex has now a state assigned, the algorithm hasn't yet converged because the edge between the vertices 0000 and 02 is not yet considered. The problematic part is that the algorithm is now required to overwrite all computation done in the right branch. This leads to a lot of messaging and bad performance.

Future Work

We could change the algorithm in a way that we do not trigger the signaling operation again after receiving a smaller rank than already known. Instead, we just store the shorter path additionally as shown in Figure 4.10 . The update process of the affected vertices is shifted to the aggregation step on the single master machine. For all cases where we have stored a smaller rank additionally, we use it to update all ranks for all vertices having the same prefix. Using a `TreeMap` this could be implemented efficiently, but still remains an overhead. We leave it as a future work to implement and assess this idea.

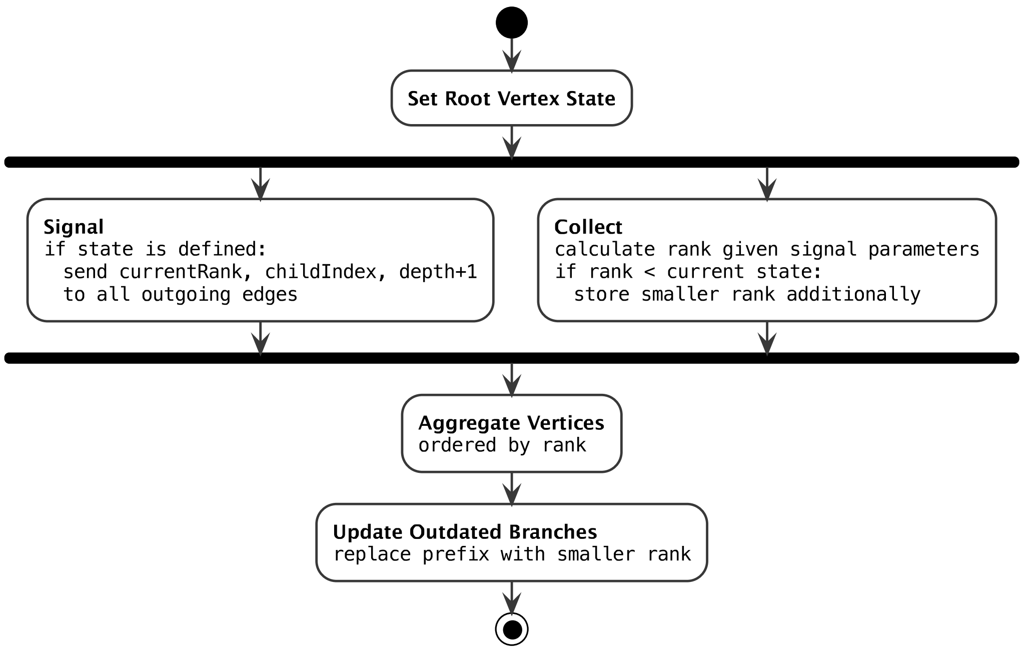


Figure 4.10: Child Indexes Queuing with Post-processing DFS Algorithm (Activity Diagram)

4.1.3 All Pair Shortest Path (APSP)

The shortest path problem searches for paths between two vertices in a graph such that the total path length is minimized. We can distinguish three different variants: the single-source shortest path problem finds the shortest paths from a source vertex to all other vertices, the single-destination shortest path problem reverses the former problem and the all-pairs shortest path problem calculates the shortest paths between every pair of vertices.

We focus our analysis on the all-pairs shortest path problem since it is the most challenging in terms of time and space complexity. The applications are for example traffic routing or centrality metrics like closeness or betweenness which we will discuss in detail in a later section.

Algorithm

Conventionally, as described by Kim et al. [2003] the problem is solved by once applying the Floyd-Warshall[2] or the Dijkstra's algorithm[3] separately for every source vertex.

At the end of the execution of the algorithm every vertex should be aware of all the shortest paths to all reachable other vertices. The idea of our algorithm, using the

[2]http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html
[3]http://mathworld.wolfram.com/DijkstrasAlgorithm.html

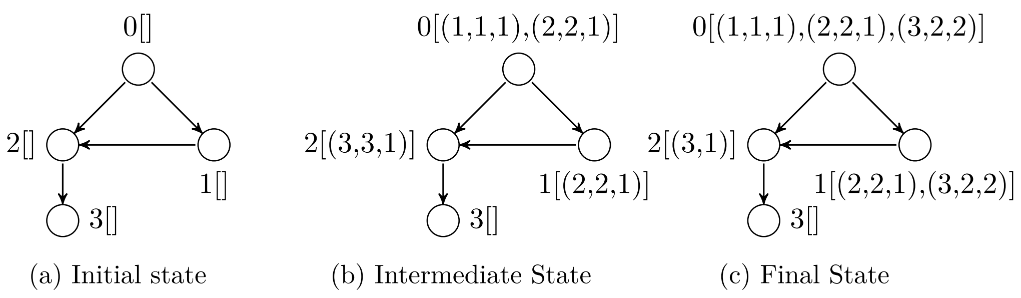


Figure 4.11: All Pair Shortest Path Computation [(targetId, nextVertex, costs)]

vertex-centric programming model, is that we start with the knowledge we have initially. Every vertex has a shortest path to itself with cost of 0. In the next step we send this information along all incoming edges to our neighbor vertices. We add the cost of the specific edge to the total costs of the individual paths. Each of the neighbor vertices checks if it is interested in the new information and potentially merges it into its knowledge. It then forwards the new information again to its connected vertices.

Figure 4.11 shows the iterative building of the shortest paths using a small directed graph. Initially, no path is known. After one step, the vertices know how to reach their neighbors and to what costs. For one shortest path leading from a source to a target vertex, the next vertex in the path and the total costs are memorized. This allows to resolve the entire shortest path. After executing the same signaling process again the final state is reached, in which every vertex knows the shortest path to all reachable other vertices.

The algorithm is both *complete* and *optimal* for graphs with strictly positive edge weights.

Implementation

Minimal Memory Footprint Keller [2014] investigated how to implement the shortest path problem using the Signal/Collect framework in order to calculate the betweenness and closeness centrality metrics. We will discuss both of them in detail later in this paper. His implementation iteratively sends all shortest paths currently known to all outgoing edges including the full path sequence. According to our evaluation, this seems to be very memory consuming and the network costs of a scaled-out server setup explode with increasing graph sizes.

Our minimal memory footprint implementation tries to address this problem. It is designed for applications and algorithms which are interested only in the costs of the shortest paths and do not require the full path sequence for every path. The computation of the closeness centrality metric would be a noteworthy example. With this restriction we have to send only the costs for a target. The total space complexity of the computation is $O(n^2)$. Assuming we represent the vertex ids as well as the costs in $32bit$ integers we have a memory footprint of $n^2 * 2 * 32bits$. For a graph with $10'000$ vertices this would

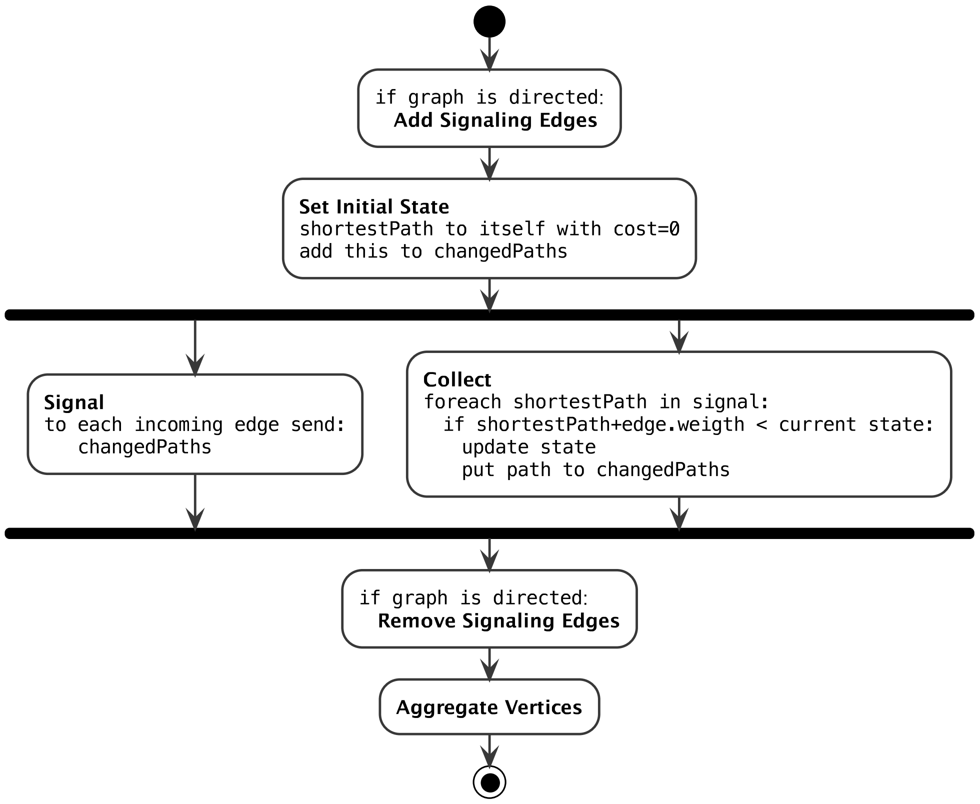


Figure 4.12: APSP Algorithm Activity Diagram

require at least 800 megabyte storage.

Figure 4.12 overviews the implementation. In Signal/Collect for a directed graph the outgoing edges are known to a vertex. Since our algorithm works in reverse order we append additional signaling edges for directed graphs to have access to the incoming edges. The implementation takes edge weights into account and allows optionally to calculate an approximated version by limiting the path length to a certain value. Again the requirement to receive every signal sent lets us build on top of the `DataFlowVertex` abstraction.

For some applications we do rely on the path sequences of the shortest paths. In the following we present three different approaches to resolve the path sequence for the shortest paths. Storing this information is very expensive since it results in a worst case scenario of $O(n^3)$. However, for many graph topologies the average path length is much smaller resulting in a more feasible footprint in practice. Referring to experiments of Milgram [1967] and the theory of Guare [1990] that everyone and everything is six or fewer steps away, assuming an average path length of 6 and again $10'000$ vertices we would end up with a total memory consumption of 2.8 gigabyte.

Pushing of Path Details The most straightforward solution is to send the entire path sequence always as a part of the signal. This has the advantage that the number of messages does not increase in comparison to the minimal approach discussed before. However, the downside is that we always send the expensive path sequences although the target vertex may not be interested in the new path, since it is already aware of a path with equal or less path length.

Eager Pulling of Path Details This issue is addressed by implementing a mechanism which allows a vertex to pull the path sequence for a target on demand. The eager pulling algorithm is implemented in the way that a vertex immediately asks for the detail information if an incoming shortest path is better than the current knowledge. The request for the path sequence is recursively passed through the sequence until one vertex is aware of the information and sends the path sequence back. This has the advantage that we can reduce the amount of sent path sequences no one is interested in. The benefit increases when the algorithm converges more and more since the degree of useful information decreases. On the other side, we introduce a lot more messages, since a request has to be sent and answered.

Lazy Pulling of Path Details The eager pulling strategy can still be wasting in the initial state of the algorithm execution. The chances that already eagerly requested path sequences get overridden during the algorithm lifecycle is high. This let us introduce another variant of the pulling strategy. The algorithm is split into two consecutive steps. Firstly, the shortest paths are computed sending and saving only their path lengths and the first vertex in the path sequence. In the second step, the paths sequences are requested and resolved recursively. The plus is that we send only path sequences which are significant. The bulk synchronous execution mode has the problem that we are forced to wait until the first step has finished completely. In a distributed system this can be a huge waste of resources when only one node is still busy and all others are idle and waiting to start the second step.

Evaluation

For the evaluation of the shortest path we use directed versions of the Barabási Albert and Erdős Rényi graphs. Since the Watts and Strogatz model is only defined for undirected graphs we use the undirected one.

In contrast to the evaluation of the other algorithms, we omit the aggregation of the calculated shortest paths in the timed section. Due to limitations of Signal/Collect, or more precise the underlying Akka[4] actor system, the aggregation of the large states is very challenging. The default settings for Akka support a maximum message size of 2.09 megabytes. If we recall the calculations we did before, for a graph with $10'000$ vertices we need to transfer at least 800 megabyte if we only want the path length for every pair of shortest paths or 2.8 gigabyte if we want to have all path sequences

[4]http://akka.io

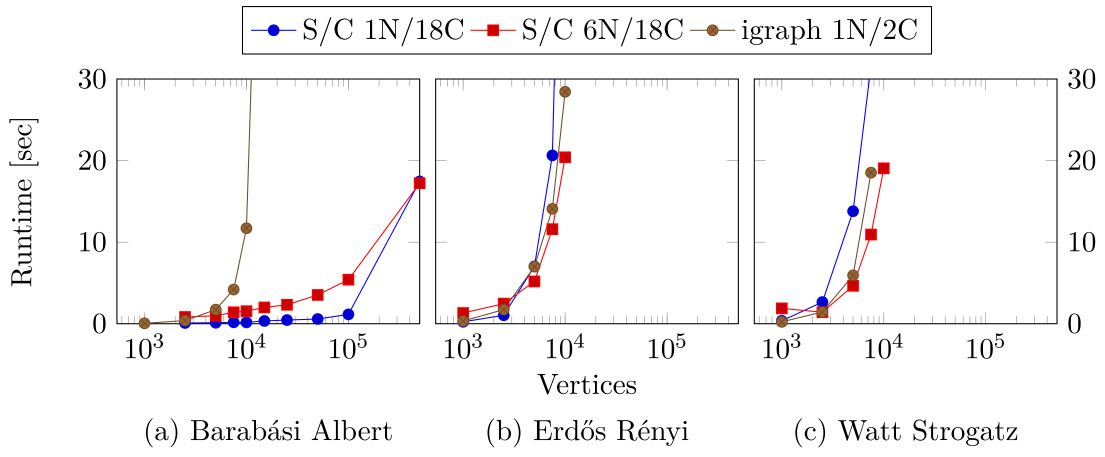


Figure 4.13: APSP Minimal Memory Footprint Scalability Evaluation (Synchronous Execution Mode)

as well. We can enlarge the value for the Akka setting, but we observe an enormous performance drop by applying it. However, for numerous algorithms one is interested in aggregated results, where you can combine the states before sending them back to the master worker. An interesting new feature in Signal/Collect would be the transparent splitting of large messages in order to support use cases where you can't combine your result to for example one single value per vertex. The evaluation of our shortest path implementations is nevertheless interesting, since there are a lot of algorithms which are built on top of the all-pairs shortest problem but eventually have a smaller result which can be transferred easily.

Minimal Memory Footprint Figure 4.13 and 4.14 plot the scalability of the algorithm using both synchronous and asynchronous execution. Please note that the number of vertices is plotted in logarithmic scale. Comparing the two modes similar conclusion as for the breadth first search can be deduced: the asynchronous execution performs worse on our multiple node setup. Even the single node configuration cannot profit, but at least it is relatively stable. We will focus the further analysis of algorithms based on the shortest path calculation on the usage of the synchronous mode. The igraph execution times are relatively constant for the different graph types. In the scale free Barabási Albert graph the Signal/Collect algorithm outperforms the igraph algorithm significantly. For the other two types the differences are smaller, but in favor of the multi node Signal/Collect cluster. The directed Barabási Albert graph scales to much larger graphs. The power-law distribution of the degrees avoids the constant neglecting of already computed shortest paths. The majority of the vertices have small in-degrees and have communicated their knowledge after a small number of steps leading to fast convergence of the entire graph. Both the Erdős Rényi and the Watt Strogatz model have a relatively uniform degree distribution resulting in much more noise.

Next, we discuss the behavior of the vertical and horizontal scaling. Figure 4.15

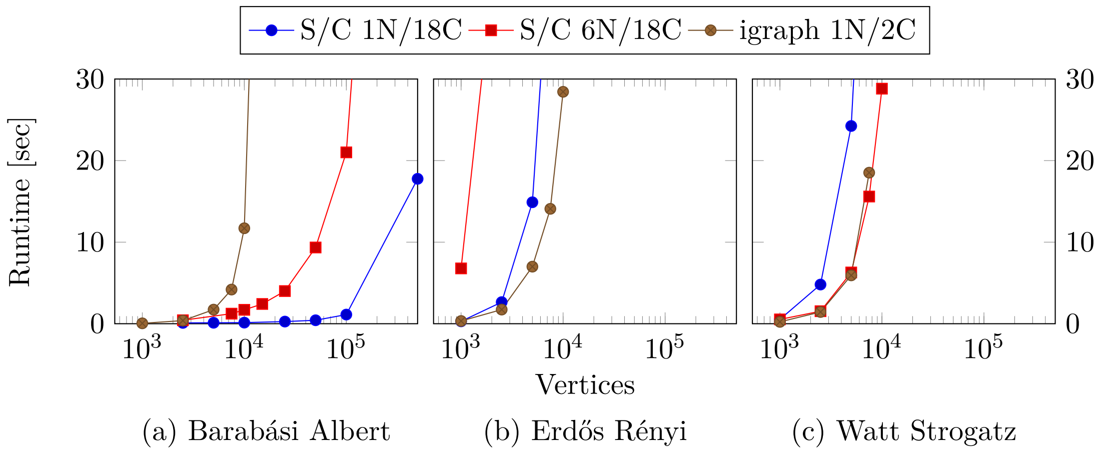


Figure 4.14: APSP Minimal Memory Footprint Scalability Evaluation (Optimized Asynchronous Execution Mode)

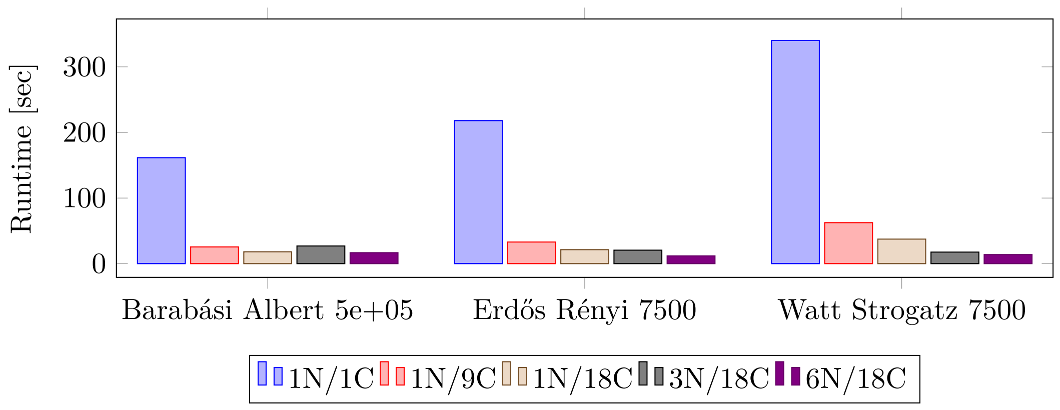


Figure 4.15: APSP Minimal Memory Footprint Parallelization Evaluation (Synchronous Execution Mode)

processed the three graphs with the server configuration analogously we have done in the BFS algorithm evaluation section. Again, we can observe that the scaling-up factor is enormous. In contrast to the BFS algorithm we can also profit by adding additional servers having increasing graph sizes. Working with 6 servers instead of 1 reduces the runtimes for the Barabási Albert with 500′000 vertices by 7.71%, for the Erdős Rényi with 7500 vertices by 44.29% and for the Watt Strogatz with also 7500 vertices by 63.19%.

To conclude, we can state that for this algorithm it is best to use the synchronous execution mode and to add as much servers as available working with large graphs. For short runtimes a single node setup may be faster.

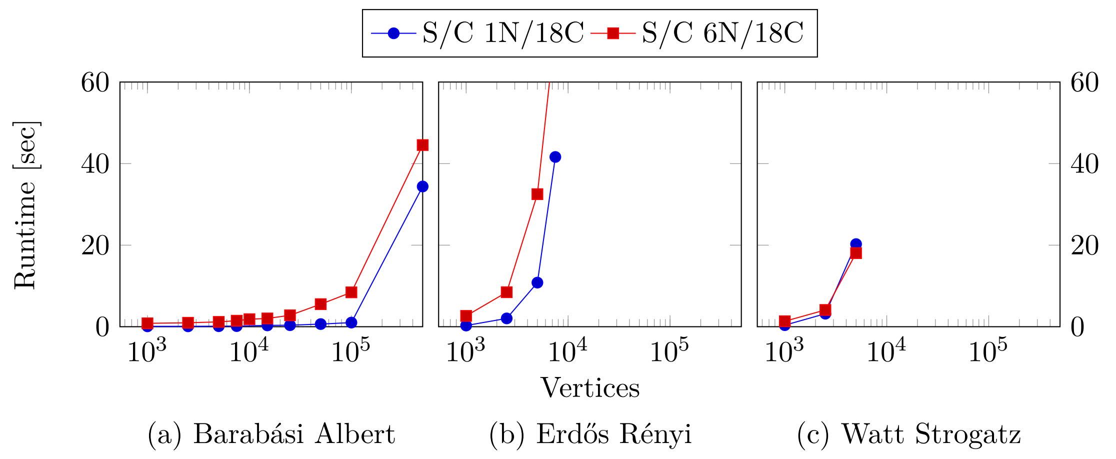


Figure 4.16: APSP Pushing of Path Details Scalability Evaluation (Synchronous Execution Mode)

Pushing of Path Details How does the performance alter when we add the full path sequence to the signal? Figure 4.16 shows a very similar trend as the minimal memory algorithm (Figure 4.15), but overall longer runtimes. For the Barabási Albert graph with 500′000 vertices the performance decrease is almost 100%. For both Erdős Rényi and Watt Strogatz only smaller graphs could be processed in a feasible time. Another observation is that the multi node configuration performed proportionally worse which is due to the higher network load caused by the bigger messages.

Eager and Lazy Pulling of Path Details Figures 4.17 and 4.18 plot the results for the eager and lazy pulling approach. Comparing them we can summarize that the lazy one performs worse than the eager one although they have a very similar behavior. Our explanation is that the synchronization costs of the two step execution mode of the lazy pulling are higher than the savings of the messages sent.

Next, we are interested in the difference between the runtimes of the eager pulling and the pushing approach. Lets consider Figures 4.16 and 4.17. The results are ambivalent. For the Barabási Albert graph the runtimes are similar, however the eager pulling algorithm could not finish the 10 million vertex graph in the single node setup. In the random graph topology (Erdős Rényi) the behavior is contrarily. While in the pushing variant the single node performs better due to the big messages, in the eager pulling approach the multi server node setup outperforms the single node one. The single node pushing algorithm performs better than the multi node eager pulling one. Looking at the last topology, the Watt Strogatz graph, there is a huge difference in favor of the eager pulling algorithm using 6 server nodes. The pushing algorithm cannot handle the sample graph with 7500 vertices. Our explanation for this phenomenon is that the eager pulling approach can profit from the shorter average path lengths of the Watt Strogatz graph. Since it asks the path sequence only for a potential shortest path this results in a better performance in this case.

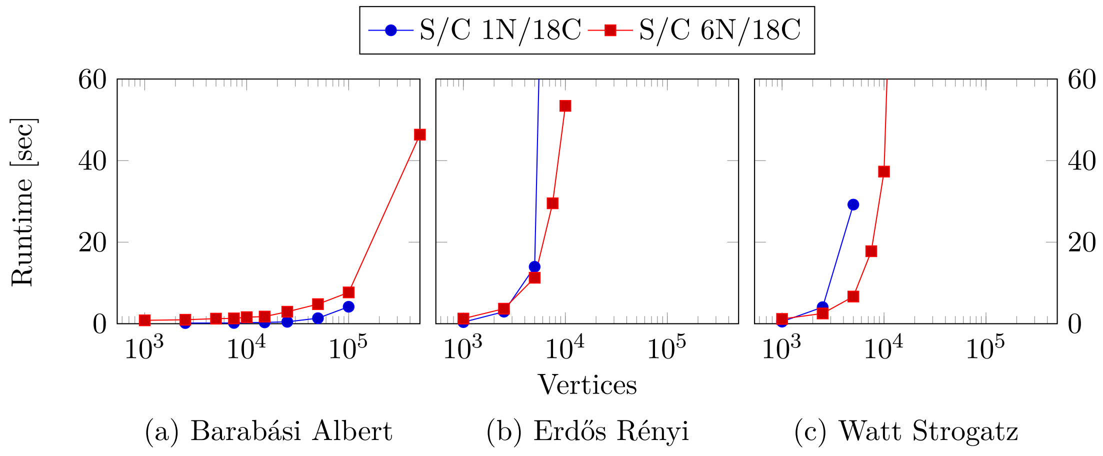


Figure 4.17: APSP Eager Pulling of Path Details Scalability Evaluation (Synchronous Execution Mode)

We summarize that the topology of the graph is crucial for both the selection of the algorithm as well as the server scaling strategy.

Future Work

Similar to the BFS we could extend the Signal/Collect framework to support pluggable message combiners on the worker nodes. This would allow us to discard paths before sending it to other nodes.

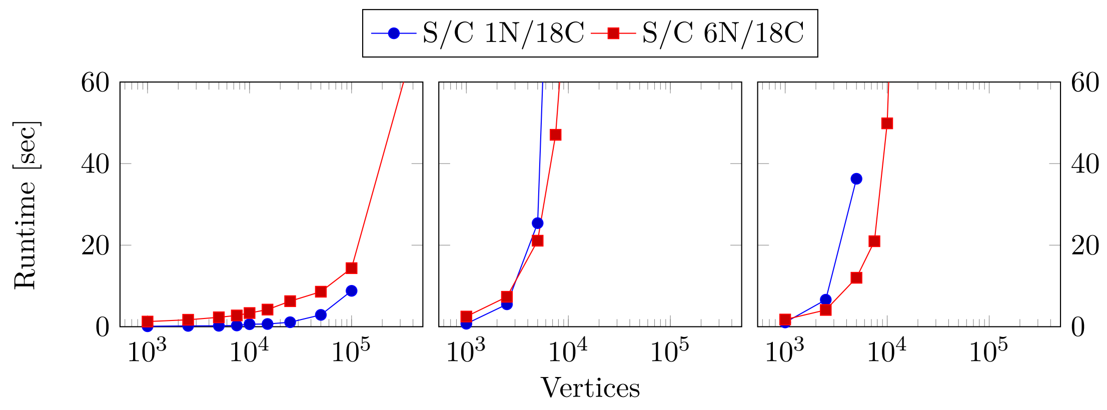
Another functionality which could have a lot of potential is the usage of a clever, algorithmic specific partitioning schema. Kalpana and Thambidurai [2014] present a technique based on degree which could be used as a heuristic to minimize the messaging between multiple nodes.

A promising approach would be to extend Signal/Collect's scoreSignal method to prioritize the signals. We argue that it could beneficial to send the signals in increasing order respect to the incoming degree of the source vertex. The chance that a vertex with a small amount of edges directing to it gets new information is lower than for a huge hub. This would reduce the noise in the system.

As mentioned before, the possibility to aggregate a large amount of data from the master worker node should be supported transparently by the Signal/Collect framework.

4.2 Graph Layout

Aesthetically pleasing visualization of a graph can be important to understand the topology of a network. Generally, layout algorithms position the vertices in a predefined space trying to minimize the number of edge crossings while keeping the edge lengths as homogenous as possible.



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.18: APSP Lazy Pulling of Path Details Scalability Evaluation (Synchronous Execution Mode)

4.2.1 A Force-Based 2-Dimensional Graph Layout Algorithm

Tutte [1963] introduces the idea of attaching forces to the vertices and edges based on their relative positions and then to use these forces to iteratively letting the system settle into an equilibrium with minimized energy.

Algorithm

Fruchterman and Reingold [1991] propose a force-based layout algorithm based on two principles: the vertices connected by an edge should be drawn near each other but the vertices should not be drawn too close to each other. For every iteration the algorithm consists of three steps: calculate the effect of attractive forces on each vertex, then calculate the effect of repulsive forces and finally limit the total displacement to some maximum value which decreases over time. Fruchterman and Reingold refer to this maximum value as temperature. If it is hot, the vertices move faster. As the layout becomes better, the temperature cools and the amount of adjustment becomes smaller and smaller. Once the vertices stop moving, the algorithm terminates. The attractive forces are computed between adjacent vertices and the repulsive forces between all pairs of vertices.

An attractive force f_a between two adjacent vertices u and v is given by

$$f_a(d_{uv}) = \frac{d_{uv}^2}{k} \tag{4.5}$$

whereby d_{uv} is the distance between the two vertices and the optimal distance k is defined as

$$k = C * \sqrt{\frac{a}{n}} \tag{4.6}$$

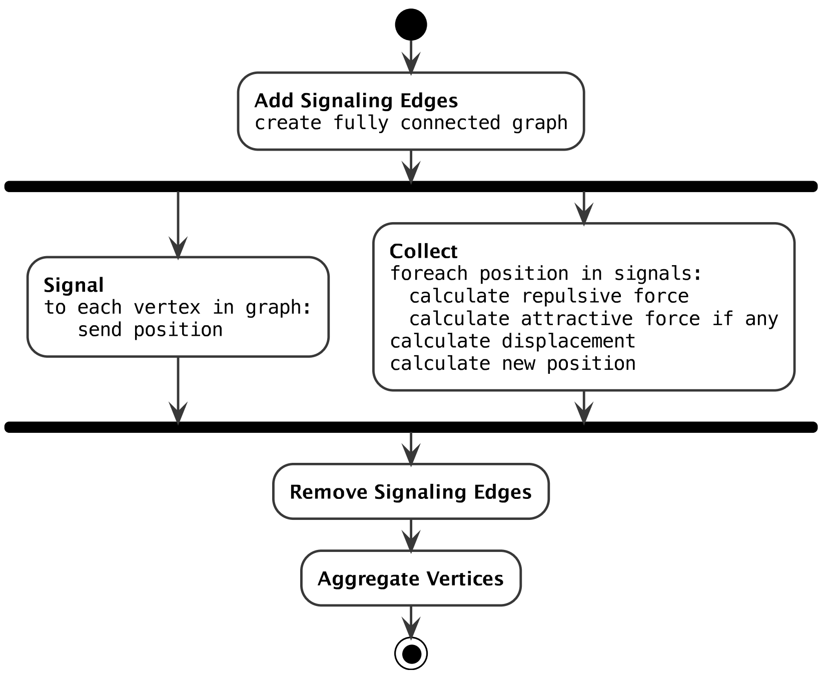


Figure 4.19: Fruchterman Reingold Algorithm Activity Diagram

given the area of frame a, the total number of vertices n and a constant C to readjust the distance between the vertices.

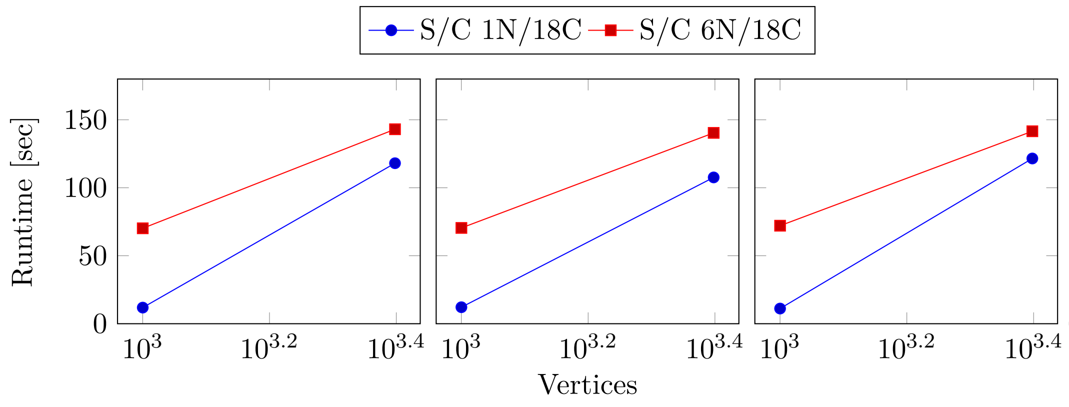
The repulsive force f_r between two vertices u and v is

$$f_r(d_{uv}) = -\frac{k^2}{d_{uv}} \tag{4.7}$$

Implementation

Figure 4.19 overviews the main steps implementing the algorithm in Signal/Collect. Due to a missing global state we are forced to create a fully connected graph in order to broadcast the positions of the vertices to each other. For each received position of another vertex the repulsive and the attractive forces (if there the vertices are neighbors) are calculated. Using the forces the displacement multiplied by current the temperature and the resulting new position are computed. For this implementation we can use the *DataGraphVertex* abstraction, which only tracks the most recent signals and triggers the collect function batch-wise. The possible drop of outdated position signals is not of importance.

In our Python Client we can call the distributed layout algorithm to get returned a pair of X- and Y-coordinates for each vertex. The call is integrated within the igraph plotting process allowing to plot the graph on the local machine transparently.



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.20: Fruchterman Reingold Scalability Evaluation (Synchronous Execution Mode)

Evaluation

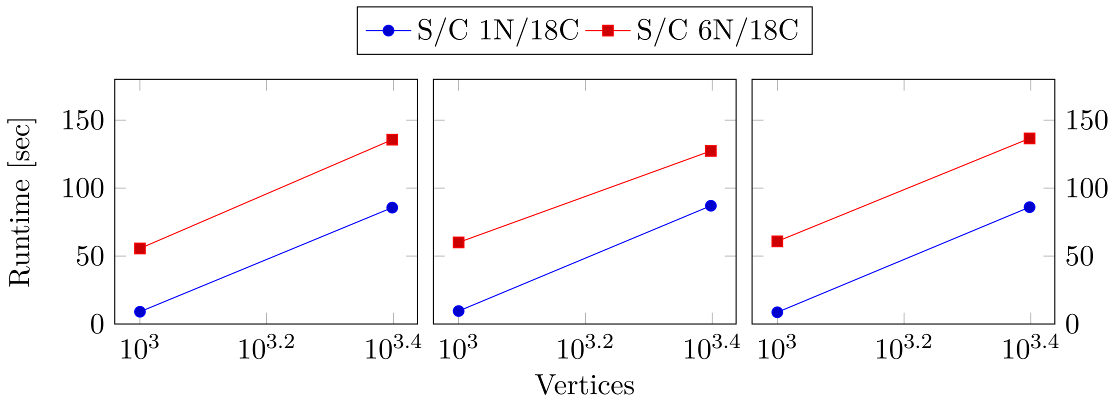
We evaluate the layout algorithm using 500 iterations. igraph does provide an implementation of the Fruchterman Reingold algorithm. However, comparing the two implementations regarding performance is difficult since the algorithm termination conditions and temperature cooling differ. For this reason we restrict the evaluation to the Signal/-Collect implementation.

Figures 4.20 and 4.21 shows the execution time for the usual graphs for the synchronous respective the asynchronous execution mode. Since we build a fully connected graph the topology of the origin graph is not a determining factor for the runtimes. For this *DataGraphVertex* algorithm the asynchronous execution outperforms the synchronous one. Especially in the single server mode.

Similar to the algorithms discussed before, the speedup using more cores on a single machine is highly beneficial as we deduct from Figure 4.22. Scaling-out drops the throughput by a factor of about three. The immense connectivity and message flow is not suitable for a server cluster configuration.

Future Work

Implementing this kind of algorithms where you rely on information from all other vertices is the counter example of the underlying idea of vertex-centric programming model. Rather than using a fully connected graph to send the positions of the vertices it could be possible to implement a grid based approximation. However, an implementation using Signal/Collect is very challenging without introducing a new bottleneck at the synchronous point which is responsible for the continuous reassignment of the vertices to new grid slots. Tamassia [2013] also describes multiscale variants as a way to deal with large graphs.



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.21: Fruchterman Reingold Scalability Evaluation (Optimized Asynchronous Execution Mode)

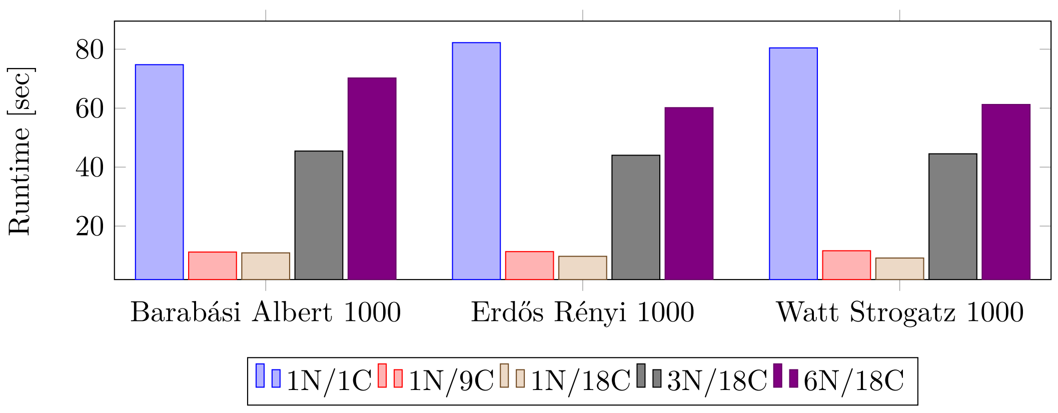


Figure 4.22: Fruchterman Reingold Parallelization Evaluation (Optimized Asynchronous Execution Mode)

4.3 Graph Transformations

In this section we discuss two transformations which return one or more new graphs from an existing one.

4.3.1 One-mode Projection of Bipartite Networks

The one-mode projection is used to compress bipartite networks, graphs which vertices can be divided into two disjoint sets and connections are only possible between two vertices of different sets. One interesting type of bipartite network is the collaboration network, which is generally defined as networks of actors connected by a common collaboration act. There are many natural appearances, as for example movie actors connected by costarring the same movie.

Algorithm

Zhou et al. [2007] give a good overview on how to project bipartite networks. Given a bipartite network with the vertex types X and Y, the one-mode projection onto X results in a network containing only X vertices. They are connected when they have at least one common neighboring Y-vertex in the original network. The simplest way is to project a bipartite network onto an unweighted graph without considering the frequency of the connection. Although some qualitative topological properties can be deducted from this approach, the quantitative loss of information is obvious. For example, the information that two actors in a collaboration network have starred in 10 movies together is treated the same as if the two had appeared on screen together. To better reflect connections, one has to use the bipartite graph to quantify the weights in the projection graph. A straightforward approach is to weight an edge by the number of times a corresponding collaboration occurs. Our implementation keeps the multiplicity of the edges as an edge attribute.

Implementation

As in the previous sections, Figure 4.23 summarize the implementation in Signal/Collect in form of an activity diagram. We can transfer all vertices to the corresponding projections in one iteration. The ideal approach would be to implement this in a distributed for each loop on the worker nodes, but that is not possible since it requires the serialization of the entire projection. Therefore, we iterate over the graph vertex by vertex and pull the vertex information and its attributes to the master worker to add the vertex to the projection of the same type. The vertex is created on the fly to keep the space complexity low. Having projected the vertices we have to distribute the new vertex ids of the projections if we want to follow the convention of igraph to have only consecutive vertex ids starting from zero. Otherwise, we can omit this step. To know the edges in the projections, for each vertex, every pairwise combination of the target vertex ids of the outgoing edges is sent to one vertex of the pair. The signal's target vertex increments

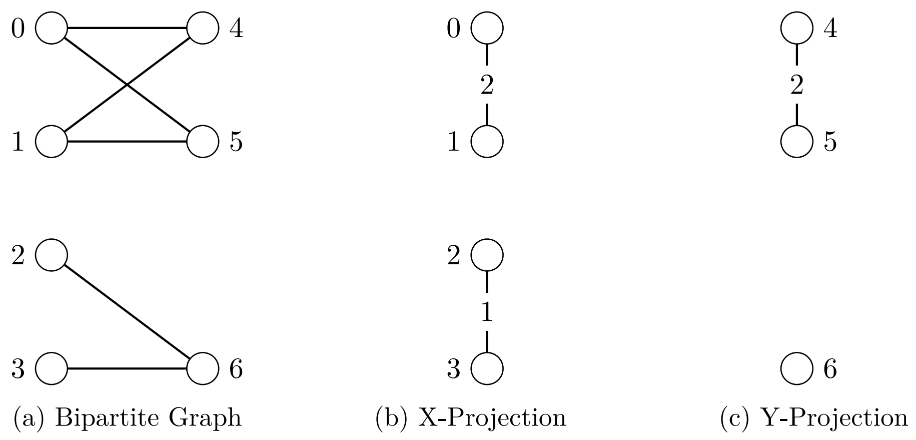


Figure 4.23: One-mode Projection of Bipartite Networks

the counter per edge pair to allow the support of the discussed multiplicity. After this step vertices of type X have the edge connections for the Y projections and vice-versa. The master worker can then iterate again over all vertices, collect the edges and build them in the projections. Finally, we remove the signaling edges to have a clean state for the execution of the next algorithm. We cannot afford the drop of any sent signals, therefore the `DataFlowVertex` is used as base.

Evaluation

Figure 4.25 shows the runtimes for the construction of the projections. We restrict the evaluation to one worker node since the Slurm Signal/Collect deployment project[5] creates only one Akka system for one execution run. A scaled-out distribution of the projections would require a unique Akka system for every created graph. We observe that the execution times are linear to the graph size. Scaling-up the server does not result in a performance drop. The synchronous code blocks dominate the execution time and neglect the parallelized parts.

Future Work

Observing the execution times we can deduct that the runtime is the larger constraint than the memory. Therefore, we may could optimize the algorithm by pulling the entire data from the distributed vertices to the master using the built-in Signal/Collect aggregation operations and then iterate over the local data. This could result in lower latencies, but enforces that the entire graph fits into the memory of the master and that the aggregation supports the transfer of large data. As we have seen aggregating the computed shortest paths, the underlying Akka messaging system is not designed to transfer huge amounts of data in one chunk.

[5]https://github.com/uzh/signal-collect-slurm

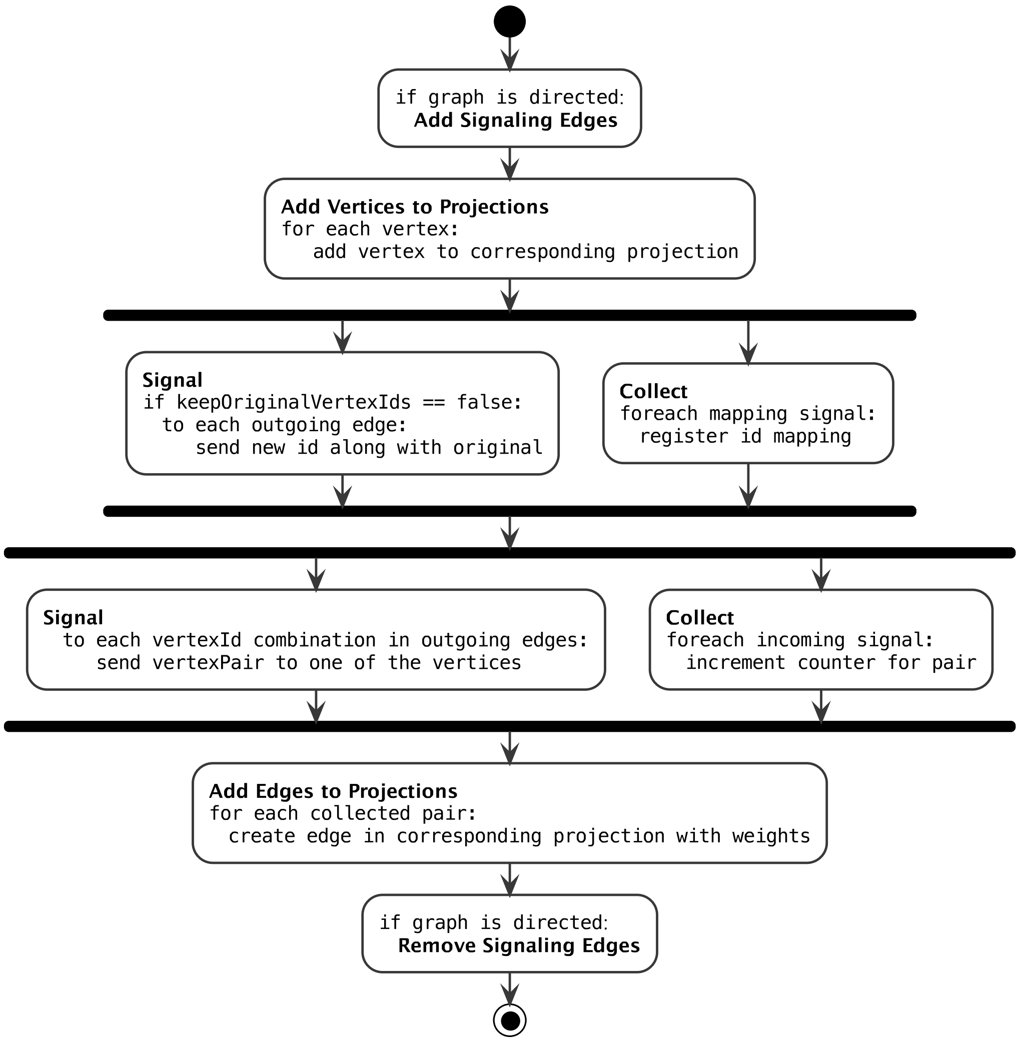

if graph is directed:
Add Signaling Edges
Add Vertices to Projections
for each vertex:
add vertex to corresponding projection
Signal
if keepOriginalVertexIds == false:
to each outgoing edge:
send new id along with original
Collect
foreach mapping signal:
register id mapping
Signal
to each vertexId combination in outgoing edges:
send vertexPair to one of the vertices
Collect
foreach incoming signal:
increment counter for pair
Add Edges to Projections
for each collected pair:
create edge in corresponding projection with weights
if graph is directed:
Remove Signaling Edges

Figure 4.24: One-mode Projection of Bipartite Networks Activity Diagram

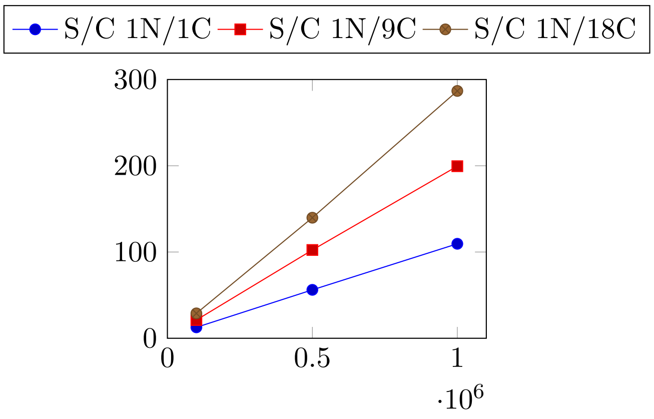


Figure 4.25: One-mode Projection Scalability Evaluation (Synchronous Execution Mode)

4.3.2 Line Graph Construction

A line graph is built from another graph representing the adjacencies between edges of the original graph. As a possible application consider a wireless ad-hoc network. Two edges cannot be simultaneously active if they are incident to the same vertex. This interference can be modeled using the line graph of the original network. More details can be found in the paper of Ganesan [2014].

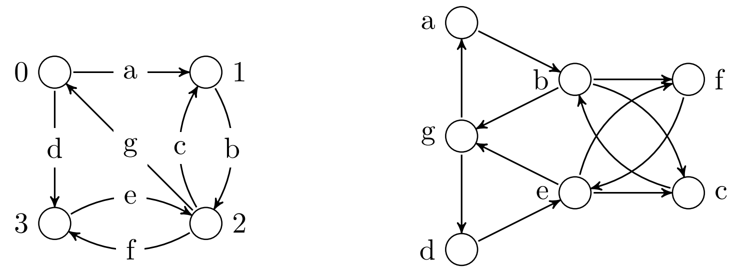
Algorithm

Referring to Weisstein [2015c] a line graph $L(G)$ of an undirected graph G is obtained by associating a vertex with each edge of the graph and connecting two vertices with an edge if the corresponding edges of G have a vertex in common. The line graph of a directed graph G is the directed graph $L(G)$, whose vertex set corresponds to the arc set of G and having an arc directed from an edge e_1 to and edge e_2 if in G, the head of e_1 meets the tail of e_2. Figure 4.26 shows an example of a directed graph with its corresponding line graph.

From the perspective of a vertex v, every pairwise combination of an incoming edge $e_{in}(v)$ and and an outgoing edge $e_{out}(v)$ in the directed graph G is an edge e in $L(G)$ pointing from the transformed vertex e_{in} to transformed one e_{out}. This formulation is already very near the implementation discussed in the next section.

Implementation

The Figure 4.27 shows the process of the construction of a line graph from an existing graph in the framework. If the graph is directed we add helper edges in reverse order to have access to the incoming edges in a vertex instance. We then iterate over every edge in the graph and put them as vertices in the line graph while transferring all attributes. In the next step we add an edge to the line graph for every combination of outgoing and



(a) Original Graph (b) Line Graph

Figure 4.26: Line Graph Constructions

incoming edges in the original graph. Since undirected graphs are modeled as double directed graphs in Signal/Collect we can reuse the same algorithm.

Evaluation

The runtime of the transformation process is independent from the topology of the graph and behaves the same for both the Barabási Albert and the Erdős Rényi graphs as shown in Figure 4.28. The Watt Strogatz sample graphs are undirected in contrast to the other two. Since in Signal/Collect an undirected graph is modeled as a doubly connected directed one, the transformation involves the creation of more edges. Keeping in mind that the x-axis is in plotted in logarithmic scale the algorithm handles an increasing number of vertices very well.

We are confronted by the same problems as discussed for the one-mode projection. There is trade-off between aggregating all data fast to the local master and then building the line graph or iteratively pulling the data and building the new graph on the fly.

4.4 Centrality Measures

Boldi and Vigna [2014] give an excellent introduction into axioms for centrality. One of the crucial question when analyzing a network is to determine which of its vertices are more central. We discuss four important classic centrality measures.

4.4.1 Degree Centrality

The degree of a vertex is the most basic centrality metric. The degree can be interpreted in terms of the chance of a vertex for observing whatever is flowing through the network.

Algorithm

Weisstein [2015d] defines the degree of a of vertex v of a graph G as the number of graph edges which touch v. For directed graphs we can distinguish between two types

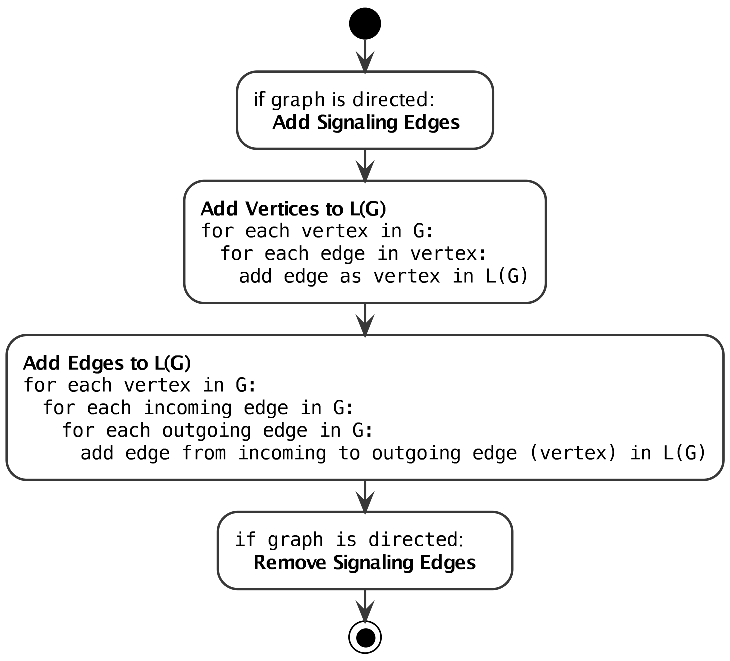


Figure 4.27: Line Graph Construction Activity Diagram

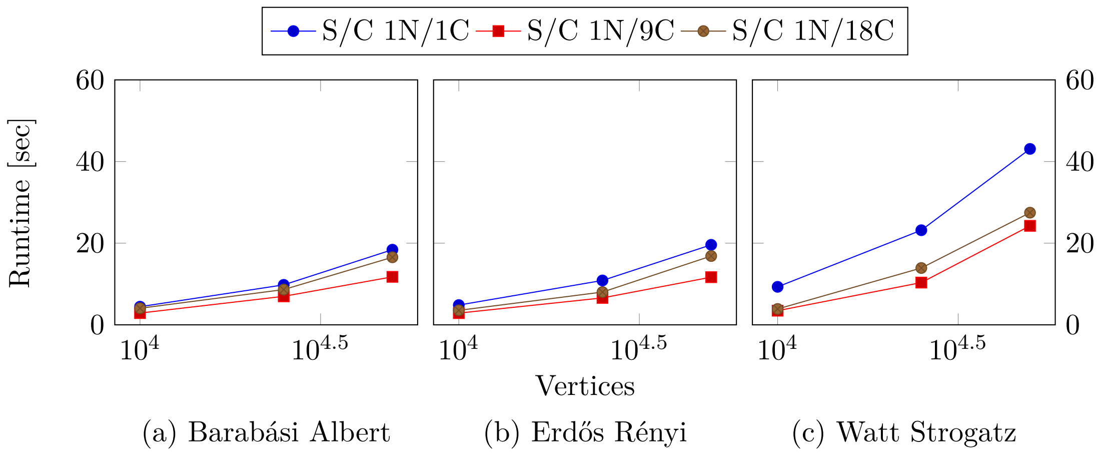
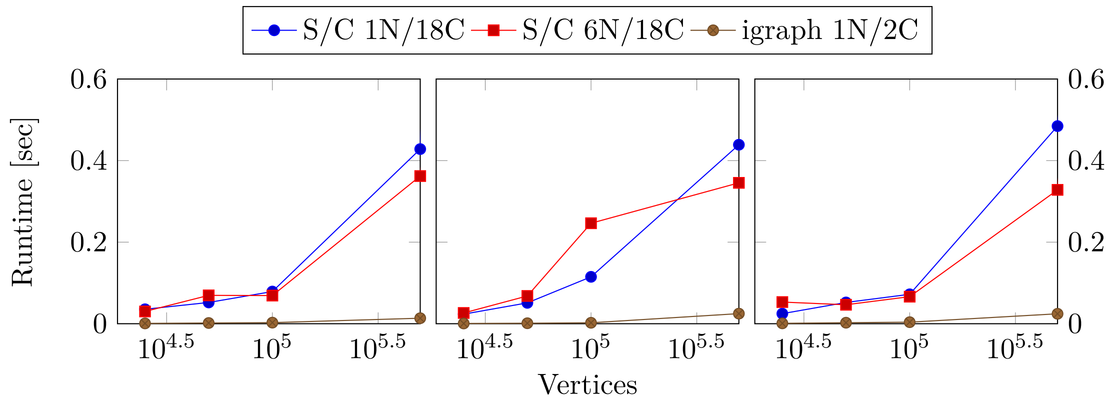


Figure 4.28: Line Graph Construction Scalability Evaluation



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.29: Degree Scalability Evaluation (Synchronous Execution Mode)

of degrees, number of incoming edges known as the in-degree and number of outgoing edges, the out-degree.

Implementation

The implementation to extract the degree values of vertices in an undirected graph or the out-degrees in a directed graph is trivial since every vertex has a list of its outgoing edges. We just have to aggregate the information. For the in-degrees in the directed graph the aggregation needs a little twist. Using the classic parallel map reduce counting pattern we put the target ids to a map and sum the values in the reduce step.

Evaluation

Figure 4.29 shows a very uniform behavior of the out-degree algorithm for all graph types with some outliners due to the very short execution times. The algorithm does not use the graph topology for the execution, but just aggregates the data from the distributed vertices in a divide and conquer approach which does benefit from a scaled-out setup. However, the local igraph instance can access the data way faster for graphs fitting in the local memory having an incredibly fast response time of 13.5 ms for a graph with 500′000 vertices. The performance of the in-degree aggregation is almost identical and we will omit its detailed discussion.

4.4.2 Closeness Centrality

The closeness centrality assigns high centralities values to vertices that have short average distances to other reachable vertices. The intuition is that the lower the distances of a vertex to all other vertices is the more central it is.

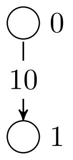
Algorithm

Bavelas [1950] defines closeness $C(v)$ of a vertex v as the reciprocal of its farness to all other vertices. This is the sum of the distances of all shortest paths between all pair of vertices.

$$C(v) = \frac{1}{\sum_{v \neq w} d(v, w)} \tag{4.8}$$

This is well defined for strongly connected graphs. In general directed or disconnected graphs the distance between two pairs of vertices can be infinite. There are several workarounds for this problem. To handle disconnected graphs one can limit the closeness measure to the largest component in the graph. However, this is no general solution for directed graphs where no link has to exist for every pair.

In igraph, the closeness implementation[6] takes the total number of vertices instead of the path length if there is no (directed) path between vertex v and w. This workaround leads to unexpected out-comings applying it to a graph with edge weights larger than 1. Figure 4.30 illustrates the igraph closeness calculations for a directed weighted graph. Although vertex 0 has a path to 1 and 1 has no outgoing paths at all, igraph calculates a smaller closeness for 1 than for 0.



(a) Sample Graph

Vertex	igraph Closeness	Harmonic
0	0.1	0.1
1	0.5	0

(b) Calculated Values

Figure 4.30: Closeness Calculation of Directed Weighted Graph by igraph

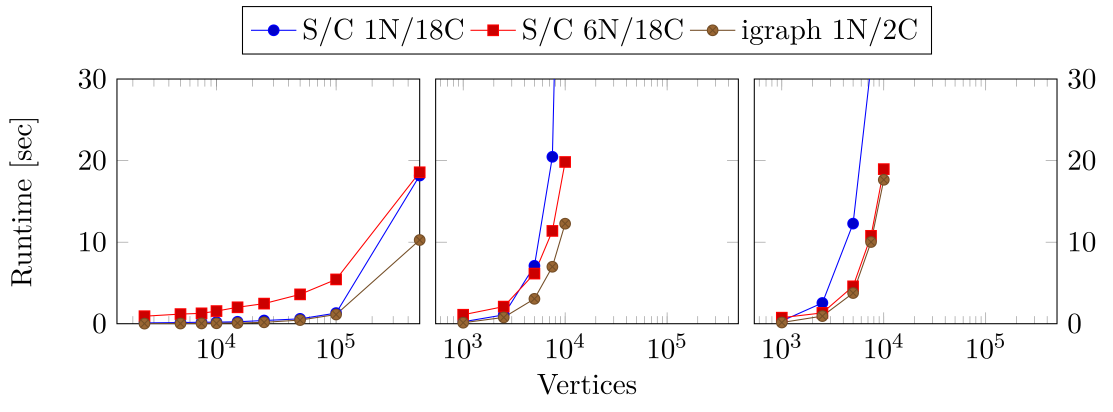
Boldi and Vigna [2014] introduce the harmonic centrality by replacing the arithmetic mean of the distance with the harmonic mean. Assuming $\infty^{-1} = 0$, the harmonic mean can handle infinite distance cleanly and can be applied to graphs that are not strongly connected. Figure 4.30 shows that vertex 1 has a closeness of 0, since it has no outgoing edges.

$$H(v) = \sum_{v \neq w} \frac{1}{d(v, w)} \tag{4.9}$$

Implementation

The calculation of the closeness metric only relies on the distances of the shortest path between all pairs of vertices. The sequence of the shortest path is not required for the computation. This allows us to reuse the Minimal Memory Footprint algorithm described in the section 4.1.3 and to change only the last aggregation step.

[6]http://www.inside-r.org/packages/cran/igraph/docs/closeness



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.31: Closeness Scalability Evaluation (Synchronous Execution Mode)

Graph Name	#Vertices	#Edges	Keller's Runtime [sec]	Our Runtime [sec]
yeast	2361	7182	14.28	1.64
facebook	4039	88234	358.19	39.94
power	4941	6594	2.23	3.67

Table 4.2: Closeness Runtime Comparison to Keller [2014]

Evaluation

The algorithm builds on top of the *APSP Minimal Memory Footprint* algorithm discussed earlier and differs only in the aggregation step. Therefore, the runtimes are almost exactly the same and the same conclusions can be drawn. Since the asynchronous mode was not competitive at all we omit to plot the results of its execution. Interesting is that the performance of the igraph implementation differs drastically for both the Barabási Albert and the Erdős Rényi graphs. While our implementation could outperform it for the computation of the shortest paths, calculating the closeness values is faster on the local igraph machine. It seems that the used algorithm is optimized for the closeness problem.

Keller [2014] has also implemented the closeness metric using Signal/Collect. Although performance comparisons have to be taken with caution, Table 4.2 compares the runtime from his evaluation with ours. As Keller's evaluation, we used three machines with 23 processors on each machine. For both the yeast and facebook graphs we could improve the performance by 89%. Which is especially important for the facebook graph, having reduced the runtime by 318 seconds. The power graph performed worse (1.44 seconds), which we think is due to the fact that we included the aggregation step into the runtime evaluation and Keller did not.

4.4.3 Betweenness Centrality

Vertex betweenness is a centrality measure of a vertex reflecting the number of shortest paths between two other vertices bridging through this vertex. Assuming that the network flow follows the shortest paths we can infer that a vertex with high betweenness has a large influence in the network flow. Freeman [1977] is generally credited with the development of the betweenness centrality measure.

Algorithm

Given the total number of shortest paths σ_{st} from vertex s to vertex t and the number of those paths that pass through v $\sigma_{st}(v)$, the betweenness centrality of a vertex $B(v)$ is defined by the following equation:

$$B(v) = \textstyle\sum_{v \neq s \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4.10}$$

Implementation

Analogous to the closeness centrality metric, calculating the betweenness requires the computation of all shortest paths between all vertices. However, for the betweenness we need to track the number of shortest paths going through every vertex. Additionally, we have to consider all shortest paths having the same distance to be able to compute the ratio of the number of shortest paths described in Equation 4.10.

As discussed in Section 4.1.3 we have implemented multiple solutions to resolve the shortest paths while keeping a low memory footprint and to avoid sending the entire path from vertex to vertex. For the calculation of the betweenness we propose a two step approach with a synchronization point in between. First, we calculate the shortest paths similar to the Minimal Memory Footprint approach, while additionally keeping the next vertex of the path in memory. In the second step, we resolve the paths and count the signals sent through the vertices.

To be able to calculate the ratio of the number of all shortest paths between two vertices and the number of shortest paths going through a particular vertex, we have to extend our Minimal Memory Footprint algorithm. The shortest paths for each vertex are stored as follows:

$$paths = tv \mapsto [(nextVertex, distance, nrPaths)] \tag{4.11}$$

We have a map with target vertex *tv* as key and a list with nested tuples. In the tuple we introduce a new information, the number shortest paths we known. Figure 4.32 shows the final states of the vertices after the first step. For illustration, only the shortest paths to the target vertex 4 are listed.

Having this information we can calculate the betweenness in the second step by iteratively sending the current betweenness value along the shortest path. The betweenness for one vertex v and target vertex t pair is calculated for an alternative a as follows:

$$B(v,t) = previousSignal(t,v) * \frac{nrPaths(t,v,a)}{totalNrPaths(t,v)} \tag{4.12}$$

Figure 4.32 shows the calculation of the betweenness starting from vertex 0 and iteratively propagating to the target vertex 4 by annotating the edges.

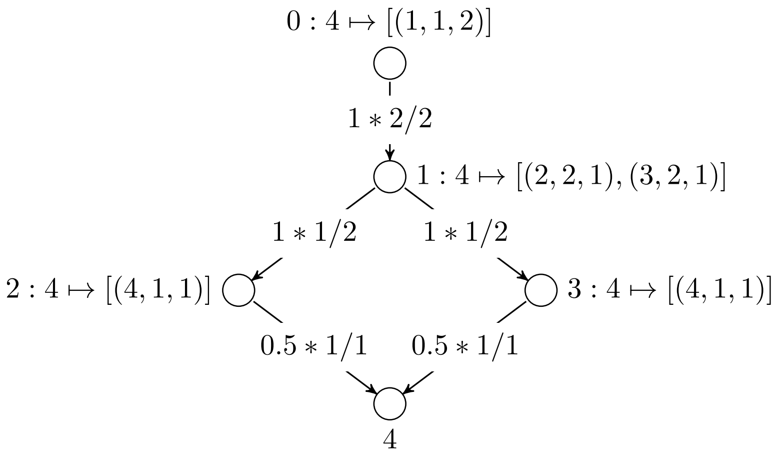

$0 : 4 \mapsto [(1,1,2)]$
$1 * 2/2$
$1 : 4 \mapsto [(2,2,1),(3,2,1)]$
$1 * 1/2$
$1 * 1/2$
$2 : 4 \mapsto [(4,1,1)]$
$3 : 4 \mapsto [(4,1,1)]$
$0.5 * 1/1$
$0.5 * 1/1$
4

Figure 4.32: Betweenness Computation (for target vertex 4 only)

Figure 4.33 illustrates the same process as an activity diagram to obtain an impression for the implementation as well.

Evaluation

Figure 4.34 evaluates the performance of the betweenness and compares it to the igraph implementation. Again, we focus on the synchronous execution mode due to its better performance. We can observe that the the runtimes are slower in comparison to the closeness by a factor of more than 2. Resolving the shortest paths in a second step to count the absolute number of paths going through the vertices is a costly operation. Additionally, finding the shortest path in the first step is more expensive since we have to keep track of paths with the same lengths. Depending on the graph topology this can lead to much more noise. This could be an explanation why the Watt Strogatz graph has the largest performance drop in comparison to the closeness implementation. Due to its topology the number of shortest paths with same lengths is higher than for the other graphs.

We planned to compare the runtimes to the ones of Keller [2014], but we realized that the algorithm is not implemented in the same way. His algorithm divides the number of shortest paths going through a vertex by the the total number of all shortest paths. However, in the standard formulation the fraction is calculated for each shortest path from a source vertex to a target vertex with equal lengths. As mentioned in the

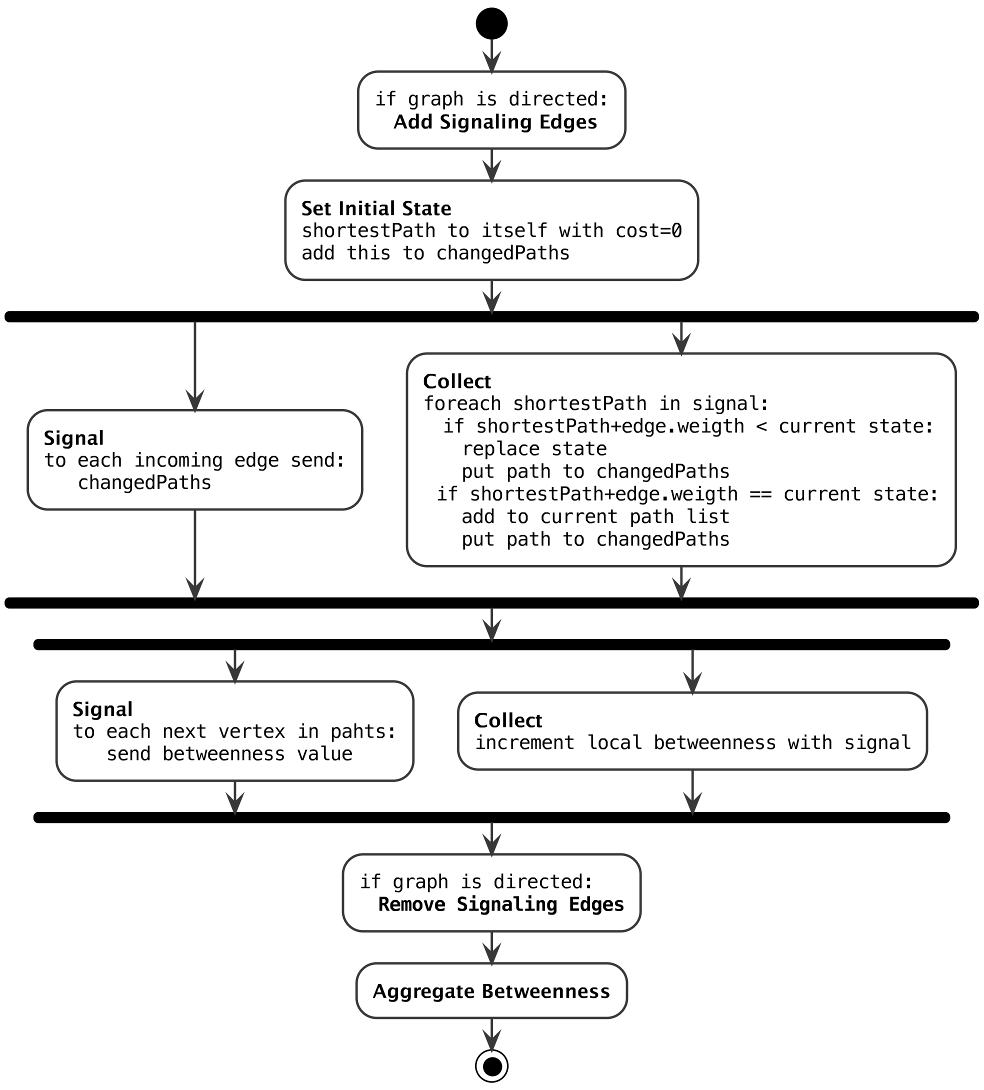


Figure 4.33: Betweenness Activity Diagram

implementation section this requires the additional tracking of paths with equal lengths which adds complexity to the problem. Therefore we omit a comparison here.

Future Work

Another approach to calculate the betweenness would be to calculate the shortest path including the path sequences as described in the APSP Section. Knowing all intermediate vertices for all shortest paths allows to calculate the betweenness value in the aggregation step. We discussed three different implementations to resolve the path sequences and pushing the path sequence always with each signal performed best. Comparing the

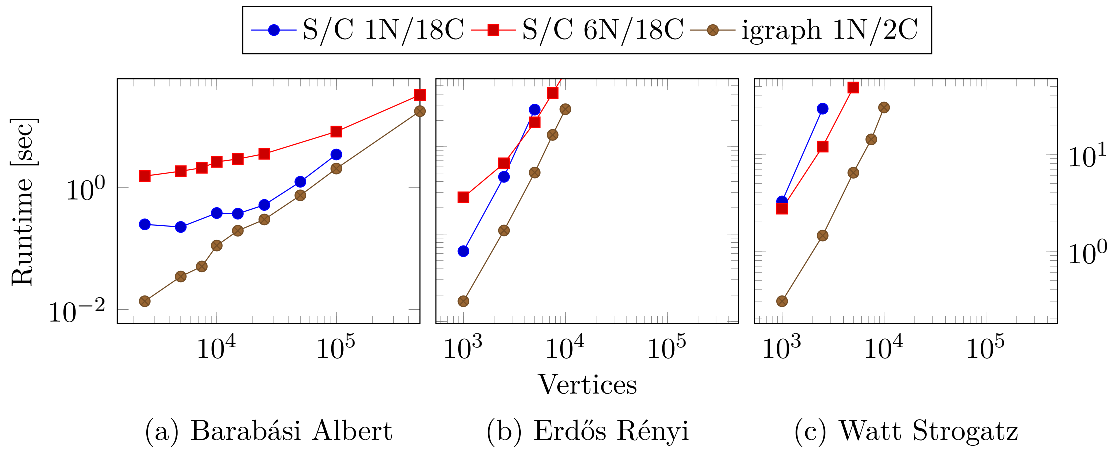


Figure 4.34: Betweenness Scalability Evaluation (Synchronous Execution Mode)

path sequence evaluation (Figure 4.16) to the betweenness one (Figure 4.34) shows that pushing the sequences is faster. This comparison is not totally fair since the shortest path algorithm does not track multiple shortest paths with the same length and does not include the additional aggregation costs, but gives an idea that the alternative approach could be feasible.

4.4.4 PageRank Centrality (Eigenvector)

With PageRank Page et al. [1999] (the founders of Google) introduce one of the most discussed and quoted centrality or importance measurement for graphs, mainly because it used to be the core of Google's web-page ranking method. The idea is that a vertex is more central if it is in relation with vertices that are themselves central. The argumentation is that the influence of some vertex does not only depend on the absolute number of its adjacent vertices, but also on their value of centrality.

Algorithm

The algorithm starts with the number of incoming edges, calculates its rank and passes its rank iteratively forward to other vertices via its outgoing edges. Hence a vertex with a high rank can endorse other vertices more strongly. This basic definition of the algorithm converges to a state where vertices without outgoing edges ultimately end up with all of the PageRank. This can be avoid this by introducing a damping factor, the probability of jumping to a random vertex, rather than by following an edge. There are various studies testing different damping factors, but the original suggestion of 0.85 by Brin and Page is the most used one.

Given the damping factor d, the set $v_{in}(v)$ representing all vertices linking to vertex v, the number of outgoing edges $e_{out}(u)$ of a vertex u then the PageRank $PR(v)$ of a vertex v is defined as

$$PR(v) = (1 - d) + d * \textstyle\sum_{u \in v_{in}(v)} \frac{PR(u)}{e_{out}(u)} \tag{4.13}$$

Implementation

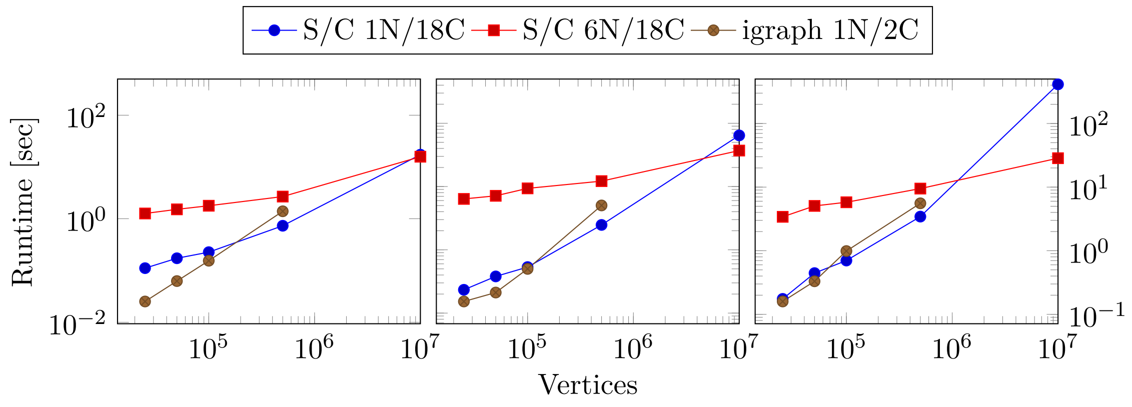
The implementation of the PageRank algorithm is very straightforward in Signal/Collect and also part of the examples in the paper of the authors of the framework Stutz et al. [forthcoming]. They propose an optimized alternative of the original implementation modeling PageRank as a data flow algorithm by signaling only the rank deltas. We adopted their implementation[7] to fit into our environment. To each outgoing edge the difference between the current state and the one of the last iteration divided by the number of outgoing edges is sent. Every incoming signal is multiplied with the damping factor and added to the current rank.

Evaluation

Stutz et al. [forthcoming] evaluate the scalability of the Signal/Collect framework with their PageRank implementation. They analyze both the vertical and horizontal scalability. However, they use different versions of the algorithm for the two evaluations and start the horizontal scaling with 4 nodes. Therefore, their paper does not enable us to compare the performance of a single node setup to a scaled-out one. As in the evaluation of the previous algorithms we would like to investigate how the runtimes change switching from a single node setup to a distributed one. Since PageRank is again an algorithm which converges towards a given threshold which differs slightly in each implementation, it is difficult to compare the runtimes to the igraph framework since the quality of the results differ. However, to get an impression of the performance we included it as a reference in the plots. As usual, Figures 4.35 and 4.36 plot the scalability using the synchronous and the optimized asynchronous mode. Please note that they are log-log plots to obtain better illustration. In the synchronous mode we can observe that the igraph implementation has worst scaling properties. We couldn't process the largest variant of our graphs containing 10 millions vertices. It is very fast for small graphs but its runtime increases fast working with larger graphs. The distributed Signal/Collect setup with 6 server nodes has the best scaling quality. The differences are immense. While for example the single node uses 412 seconds for the Watt Strogatz graph with 10 millions vertices, the distributed one requires 28 seconds which is about 93% faster.

Next, we analyze the optimized asynchronous execution mode. The runtimes for the single node setup are faster. For 10 millions vertices the performance increase is 6% for Barabási Albert, 38% for Erdős Rényi and 71% for Watt Strogatz. On the other hand, the distributed server setup has a massive performance drop: 1586% for Barabási Albert, 521% for Watt Strogatz. The random graph topology is affected most and for the 10 million vertex graph the computation could not be finished due to a buffer overflow.

[7]https://goo.gl/BLjwbp



(a) Barabási Albert (b) Erdős Rényi (c) Watt Strogatz

Figure 4.35: Page Rank Scalability Evaluation (Synchronous Execution Mode)

Figure 4.37 presents the usual parallelization plot using synchronous execution mode. The vertical scaling is beneficial as for the most of the algorithms we discussed. Looking at a graph size of 10 millions vertices using 6 servers outperforms the single node for all three graph types. Adding only 3 servers has a mixed output. For the Barabási Albert graph there is a performance drop of 32%, for the Erdős Rényi topology the runtimes are relatively constant with an improvement of 6%. Watt Strogatz profits most with an decrease runtime of 88%.

We can conclude our evaluation with the confirmation of the good scaling properties claimed in the paper of Stutz et al. regarding the PageRank algorithm.

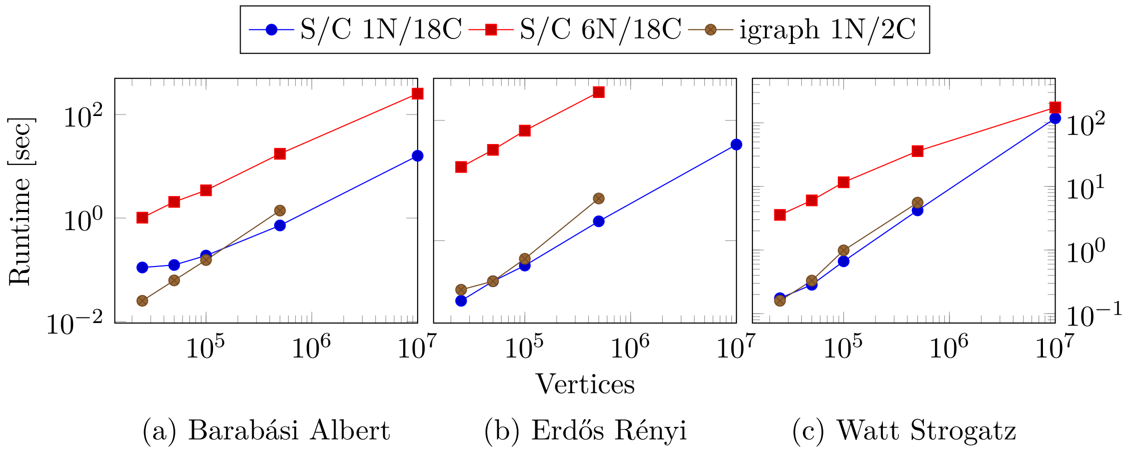


Figure 4.36: Page Rank Scalability Evaluation (Optimized Asynchronous Execution Mode)

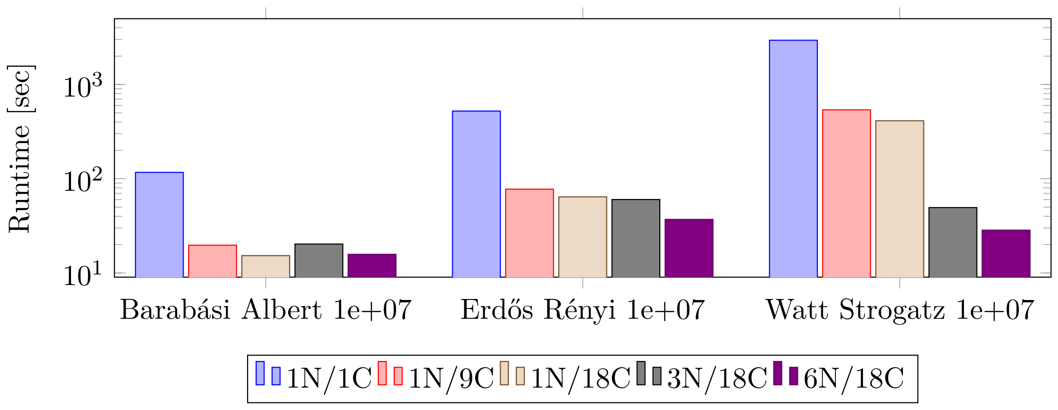


Figure 4.37: PageRank Parallelization Evaluation (Synchronous Execution Mode)

5

Limitations and Future Work

The evaluation shows that the current implementations are capable to solve some of the standard social network analysis algorithms, but struggles with others. In this chapter we discuss the limitations we are confronted with together with a suggestion of future work on how these limitations could be overcome.

5.1 Implementation

We are certain that the performance of our implementations could be improved by optimizing the code. The memory footprint of a signal is crucial for the total runtime. Using data structures which can be serialized by Kryo[1] efficiently in both time and memory aspects could lead to big improvements. However, this can lead to a worsening of the architecture and code quality in terms of readability and reusability.

The deployment of the framework to a cluster is not yet designed finally. This includes for example the programming of a user and resource management system, or the dynamic maintenance of the Akka actors for the Signal/Collect framework.

Discussing the algorithms we have already mentioned two techniques which could be beneficial. An algorithm specific *message combiner* on a server node level could decrease the number of messages sent across the network by aggregating or simply dropping them.

The computation cost is driven by communication overhead sending messages between the distributed servers. The goal is to maximize the locality of processing. Using a custom *partitioning mechanism* rather than a random hash function could help. However, finding a general graph partitioning strategy for a framework which should execute multiple algorithms on the same graph without the need of a constant reshuffling of the data is an inherently challenging task.

5.2 Programming Model

We agree with the conclusions of Elshawi et al. [2015] that parallelizing graph algorithms while preserving an efficient performance is very challenging as we have seen comparing

[1]https://code.google.com/p/kryo/

the runtimes to the serial implementations of igraph. The usage of a scalable graph framework supports the resilient, transparent distribution of the data and processing power with the price of a restrictive programming interface as for example the vertex-centric one which works well for a huge set of iterative algorithms. But many well-elaborated algorithms with extremely efficient implementations rely on having a global state or operate on adjacency matrix representations of the graphs and therefore cannot be ported trivially to a distributed environment while still providing high-performance. It is therefore difficult to reuse existing work.

Sending signals asynchronously along the outgoing edges to the adjacent vertices, as the model propagates, leads to an execution flow similar to breadth-first traversal. We have the impression that we can elegantly implement algorithms which somehow base on a BFS approach as for example PageRank or APSP, but have major difficulties when they differ. We think that for the development of a new algorithm it would be crucial to think more outside the box and to consider a rigorous reconnecting of the graph to obtain the results and to avoid to be restricted to a BFS distribution pattern. However, this is conceptually very complex for the developer.

The vertex-centric programming model divides input graphs into partitions and hides low level optimizations generally. Tian et al. [2013] introduce the *graph-centric* paradigm. It adds the partitioning mechanism as integral aspect of the programming model to allow algorithm specific optimizations. Signal/Collect does already support this from a technical perspective, but it is not part of the core programming model to support for example the dynamic physical reshuffling of the vertices.

Yan et al. [2014] propose a *block-centric* graph-parallel abstraction, called Blogel. Their claim is that it is conceptually as simple as the vertex-centric one, but works in coarser-grained graph units called blocks. Here, a block refers to a connected subgraph of the graph, and message exchanges occur among blocks. The argumentation is that the vertex-centric model largely ignores the characteristics of real-world graphs in its design and hence suffers from severe performance problems. Their model should allow an efficient partitioning reducing communication costs. We could imagine that a programming model which deals with a larger neighborhood could be very interesting for developing algorithms which do not fit perfectly into the vertex-centric design due to its expensive costs to obtain an extended view.

5.3 Signal/Collect

Signal/Collect is a very powerful platform to develop scalable graph processing algorithms. In this section we would like to notice some aspects which from our perspective could be improved. For some of them there may exist elegant solutions which we are not aware of.

Generally, the programming interface of Signal/Collect is relatively static using the reuse by inheritance principle for many parts. Although the framework perfectly allows to configure its usage down to the underlying Akka framework, the entire architecture is built to be configured at compile time. This works great if you want to execute one algorithm per distributed graph. We think the framework would be much more flexible if it would support its configuration at runtime, for example through the consequent usage of the strategy pattern. This includes the partitioning mechanism, message combiner, and of course the vertex and edge implementations. With the PluggableVertex we implemented a first step in this direction.

Another issue is the aggregation of large vertex states. The current implementation suffers from the support to transparently split the serialized aggregated data structures into chunks to send them back to the master worker.

Having the possibility to prioritize the signals could improve the converging speed of algorithms especially when combined with the asynchronous mode and the DataGraphVertex behavior which allows to drop out-dated signals. This could be implemented within the same interface using the `scoreSignal` value.

5.4 Standardized Benchmarks

Elshawi et al. [2015] mention that benchmarks need to play an effective role in helping users to make decisions regarding the choice of adequate platforms for their requirements. Designing a meaningful benchmark is challenging. At the current state, most of the reported benchmarking studies have been self-designed, including the ones in this paper, because there is a lack of standard benchmarks that can be used.

6 Conclusions

We used Apache Thrift to define a network analysis toolbox (NAT) service based on a subset of the igraph library to simplify the integration of graph processing frameworks into existing analytics environments.

We built a flexible and generic framework on top of Signal/Collect which implements the NAT interface and provides a Thrift server. The system supports the creation and maintaining of multiple Signal/Collect graph instances including garbage collection. In contrast to the plain Signal/Collect programming interface, we added the support of the consecutive execution of different algorithms on the same graph without the need to redistribute the data again. Edges and vertices can be associated with attributes.

We transferred well-known algorithms in the social network analysis domain to the vertex-centric programming model. All of them deliver correct outputs while their performance efficiency vary as we have seen in the evaluation chapter.

Our Python client allows the transparent forwarding of the implemented igraph functionalities to the Signal/Collect graph representations.

While developing the system the following insights emerged:

Influence of the Graph Topology We are surprised by the explosion of the space and time complexity for many algorithms. We can observe that the runtimes of our implementations depend much more on the topology of the graph rather than on the serial counterparts of the igraph implementation do. This is due to the fact that the number of messages sent depends often on the topology and the communication costs dominate.

Execution Mode One of the distinguishing features of Signal/Collect is the asynchronous execution mode. However, in our implementations its performance was mixed for one node setups and often poor for multi node clusters. Of course our implementations and configurations can be optimized drastically, but we argue that the complexity of developing asynchronous algorithms do often not reflect the performance benefit over a bulk synchronous system for the many applications. The conclusion is that there is not one best execution mode but the correct decision depends on the graph topologies, the server environment and the algorithms.

Scaling Strategy Developing an algorithm it is very difficult to predict its runtime on a distributed environment. This is not limited to the Signal/Collect framework. What scalability strategy is the best? We observed that vertical scaling always results in faster runtimes and can be applied without risking slower execution times due to the increased overhead. Our evaluations show that it is very difficult to estimate if an algorithm profits from horizontal scaling. In our evaluation, we used a high-end cluster with many cores per node and a very fast InfiniBand network. Nevertheless, adding more nodes did often not result in faster computations. This could be much worse using commodity hardware with slower network connections. Adaptive techniques which transparently optimize the degree of horizontal scaling depending on the characteristics of the underlying graph topology and algorithm used would be a very interesting field of research. Generally, we can conclude that it requires a numerous amount of servers to neglect the overhead of the scaled-out setup and to outperform a single node server.

Performance Comparsion to igraph Having evaluated the framework and having it compared to the runtimes of local igraph instances we come to the following conclusions. We distinguish three different cases:

The first scenario is that our Signal/Collect algorithm scales better than the igraph implementation resulting in a faster runtime given a certain graph size. This is for example the case for the all pair shortest path problem where we are only interested in the path lengths.

A second case is that Signal/Collect cannot outperform the serial implementation. But after reaching the memory limit of the local resources the Signal/Collect algorithm can still solve the problem within a feasible time. This applies for problems where the space complexity of the problem increases faster than the computation time complexity. For these cases we can profit from the resources available in a distributed system. The degree centrality metric is an example since the storage of a huge graph is more limited in a local environment although the computation of the metric itself does not require complex computations.

In the last category are algorithms for which the Signal/Collect framework again performs worse than the serial implementation, but the runtimes explode to infeasible times before reaching the memory limit. In other terms the time complexity neglects the space complexity. In this case we cannot profit from having more memory available. One approach is to deviate from the standard implementations and find other algorithms which approximate a solution for the problem. The betweenness centrality metric is an example for this category.

References

Akhter, S. and Roberts, J. (2006). *Multi-core programming*, volume 33. Intel press Hillsboro.

Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47.

Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. (2013). Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 20. ACM.

Avery, C. (2011). Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara.*

Bavelas, A. (1950). Communication patterns in task-oriented groups. *Journal of the acoustical society of America.*

Boldi, P. and Vigna, S. (2014). Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262.

Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9.

Elshawi, R., Batarfi, O., Fayoumi, A., Barnawi, A., and Sakr, S. (2015). Big graph processing systems: State-of-the-art and open challenges. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pages 24–33. IEEE.

Erdös, P. and Rényi, A. (1960). {On the evolution of random graphs}. *Publ. Math. Inst. Hung. Acad. Sci*, 5:17–61.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41.

Fruchterman, T. M. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11):1129–1164.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software.* Pearson Education.

Ganesan, A. (2014). Performance of sufficient conditions for distributed quality-of-service support in wireless networks. *Wireless networks*, 20(6):1321–1334.

Guare, J. (1990). *Six degrees of separation: A play.* Vintage.

Himsolt, M. (1997). Gml: A portable graph file format. *Html page under http://www. fmi. uni-passau. de/graphlet/gml/gml-tr. html, Universität Passau.*

Kalpana, R. and Thambidurai, P. (2014). A speedup technique for dynamic graphs using partitioning strategy and multithreaded approach. *Journal of King Saud University - Computer and Information Sciences*, 26(1):111 – 119.

Keller, F. (2014). Social network analysis with signal/collect.

Kim, H.-S., Lee, J.-H., and Jeong, Y.-S. (2003). Method for finding shortest path to destination in traffic network using dijkstra algorithm or floyd-warshall algorithm. US Patent 6,564,145.

Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J. (2014). Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041.*

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. ACM.

McCune, R. R., Weninger, T., and Madey, G. R. (2015). Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing. *CoRR*, abs/1507.04405.

Milgram, S. (1967). The small world problem. *Psychology today*, 2(1):60–67.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

Russell, S. and Norvig, P. (1995). Artificial intelligence: a modern approach.

Simmen, D., Schnaitter, K., Davis, J., He, Y., Lohariwala, S., Mysore, A., Shenoi, V., Tan, M., and Xiao, Y. (2014). Large-scale graph analytics in aster 6: bringing context to big data discovery. *Proceedings of the VLDB Endowment*, 7(13):1405–1416.

Strebel, D. (2013). Scalable forensic transaction matching: and its application for detecting patterns of fraudulent financial transactions. Master's thesis.

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. Springer.

Stutz, P., Strebel, D., and Bernstein, A. (forthcoming). Signal/collect: Processing large graphs in seconds.

Sumaray, A. and Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 48. ACM.

Tamassia, R. (2013). *Handbook of graph drawing and visualization.* CRC press.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204.

Tutte, W. T. (1963). How to draw a graph. *Proc. London Math. Soc*, 13(3):743–768.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442.

Weisstein, E. W. (2015a). Breadth-first traversal. http://mathworld.wolfram.com/Breadth-FirstTraversal.html. Visited on 30/08/15.

Weisstein, E. W. (2015b). Depth-first traversal. http://mathworld.wolfram.com/Depth-FirstTraversal.html. Visited on 30/08/15.

Weisstein, E. W. (2015c). Line graph. http://mathworld.wolfram.com/LineGraph.html. Visited on 30/08/15.

Weisstein, E. W. (2015d). Vertex degree. http://mathworld.wolfram.com/VertexDegree.html. Visited on 30/08/15.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM.

Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992.

Zhou, T., Ren, J., Medo, M., and Zhang, Y.-C. (2007). Bipartite network projection and personal recommendation. *Physical Review E*, 76(4):046115.

List of Figures

List of Tables