Universität Zürich^{UZH}

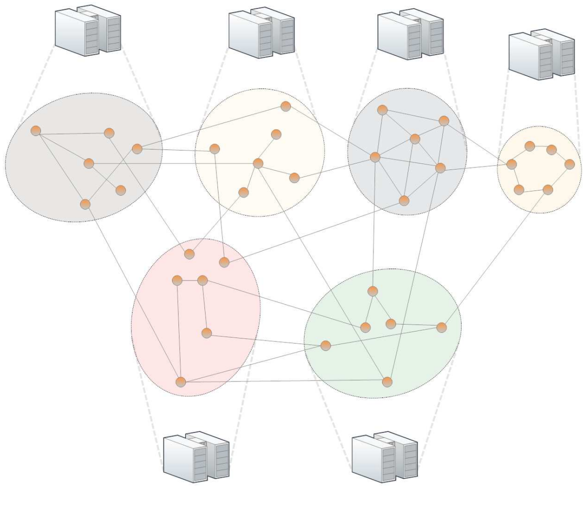
Dynamic and Distributed Information Systems

Diploma Thesis August 24, 2011

Distributed Signal/Collect

Francisco de Freitas
of Belo Horizonte MG, Brazil

Student-ID: 09-734-062
favdefreitas@gmail.com



Advisor: **Philip Stutz**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank my supervisor Philip Stutz for the frequent and fantastic help and advices throughout the duration of this work that permitted me to contribute to a highly exciting topic.

To my parents Leila and Waldo for the unconditional help and for the ”cheering”, even though from a far distance, necessary for the conclusion of my studies.

My special gratitude to Barbara, Arthur and Ellen for the unprecedented aid and care in all matters throughout these two years.

I would also like to thank Prof. Bernstein for the opportunity to work on such an interesting research subject.

Abstract

New demands for analyzing and working with large data sets establish new challenges for computation models, especially when dealing with Semantic Web information. Signal/Collect proposes an elegant model for applying graph algorithms on various data sets. However, a distributed feature for horizontally scaling and processing large volumes of data is missing. This thesis analyzes existing graph computation models and compares distributed message- passing frameworks for proposing an integrated Distributed Signal/Collect solution that tries to solve the problem of limited scalability. We successfully show that it is possible to implement distributed mechanisms using the Actor Model, although with some caveats. We also propose future works in an attempt to further enhance our solution.

Zusammenfassung

Die Nachfrage, grosse Datensaetze zu analysieren und zu bearbeiten, stellen Berechnungsmodelle vor neue Herausforderungen, besonders wenn man mit Informationen aus dem Semantic Web arbeiten muss. Signal/Collect bietet ein elegantes Modell, um mittels Graphalgorithmen verschiedene Datensaetze zu bearbeiten. Das Modell ist jedoch nicht komplett, da die horizontale Skalierbarkeit fuer die Bearbeitung grosser Datenmenge nicht offeriert ist. In dieser Arbeit werden bereits bestehende Graphberechnungsmodelle analysiert und Frameworks fuer verteilte Systeme mit Nachrichtenuebertragung verglichen, um eine integrierte Loesung fuer Distributed Signal/Collect zu offerieren. Die Arbeit zeigt auch, dass es trotz einigen Vorbehalten moeglich ist, verteilte Mechanismen mit dem Actor-Modell erfolgreich zu integrieren. Wir empfehlen, zukuenftig weitere Forschungsarbeit zu betreiben, damit unsere Loesung weiter verbessert werden kann.

Table of Contents

1 Introduction

Today's demand for analysis of Semantic Web Data is growing at a fast pace, mostly due to the increase of data being available on the Web. Every day, millions of pages are published on the Internet [ILK, 2011] (refer to Figure 1.1) and models for working with large volumes of data are not abundant.

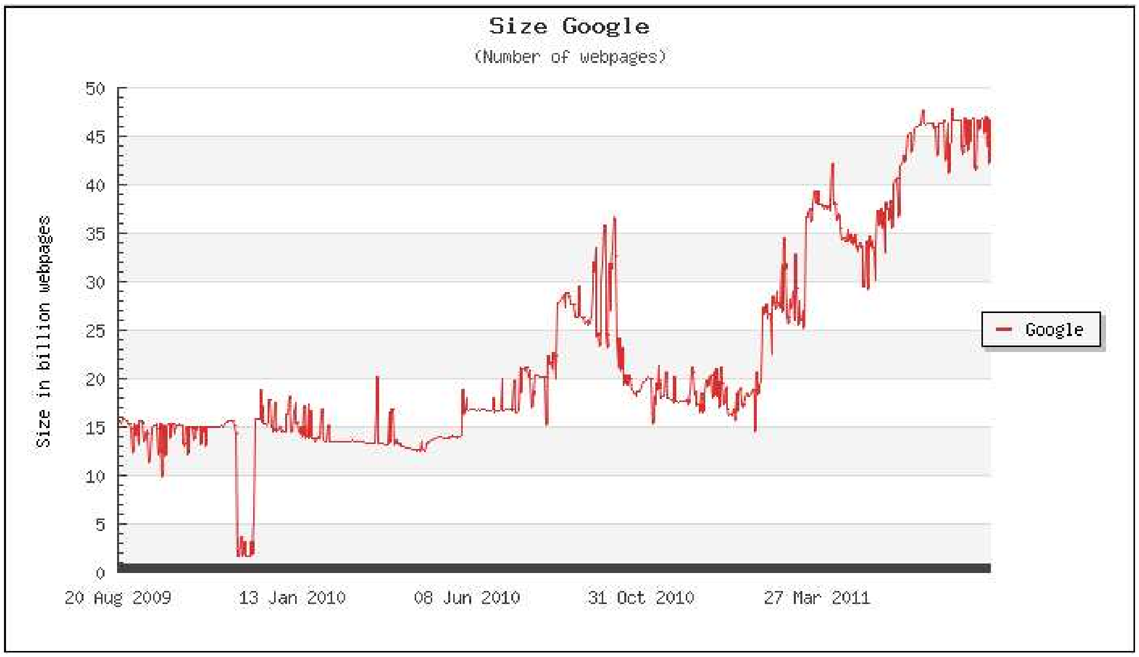


Figure 1.1: Estimated Google web-graph size

Programming models such as MapReduce [Dean and Ghemawat, 2008] have proven to work well for some graph algorithms, but most of the problems require a rather cumbersome approach to implement the solution finding algorithm. Also worth noticing is Google Pregel which has shown to scale to use massive graphs, although this framework introduces some limitations in its computation model [Malewicz et al., 2009] . The Horde framework, even though it has a somewhat simplistic approach towards a large scalability solution, shows that it is possible to achieve better results than frameworks like MapReduce [Xu et al., 2009]. GraphLab [Low et al., 2010] is an intriguing attempt in the field of parallel graph processing that proposes a model for solving machine learning problems in a new manner by combining machine learning processing with parallel computations in graph-like algorithms.

In light of the scarcity of available tools, Signal/Collect proposes a solution that is concise and also offers a sophisticated model for graph processing. Its design has a model that fits well for a satisfactory number of graph algorithms. Therefore, Signal/Collect provides a set of tools that allows accomplishment of tasks in a performing way when compared to other paradigms, such as those presented before [Stutz et al., 2010]. However, Signal/Collect is limited to a shared-memory execution only. In other words, it does not scale horizontally which means that it cannot work in a distributed way. For this reason, it lags behind other programming models in terms of features.

Cluster and cloud computing are increasing in availability by observing the offering of services from Google [1], Amazon [2] and IBM [3]. They are listed amongst the biggest players to offer commercial services for large scale processing, provided that more and more data needs to be processed using distributed resources. Therefore, large scale systems are an advantage for the fact that they can enable the loading of huge amounts of data for analysis. Consequently, having such systems that deal with a considerable volume of data is becoming a standard.

1.1 Motivation

As stated in the introduction, usage of scalable systems is becoming a consolidated trend. Nevertheless, dealing with huge datasets can become ineffective if one does not have the right tools for correctly perform processing and analysis. Having said that, one must not forget that the Signal/Collect implementation, prior to the conclusion of this work, does not scale horizontally,

[1]http://www.techno-pulse.com/2011/04/google-cloud-computing-services.html
[2]http://aws.amazon.com/ec2/ - Amazon Elastic Compute Cloud (Amazon EC2)
[3]http://www.ibm.com/cloud-computing

somewhat limiting the execution to rather small graphs. On the other hand, since Signal/Collect proved to have good performance and great potential for efficiently performing graph-like computations, it is more than reassuring that the framework is open for improvements to make it perform even better.

Making Signal/Collect distributed poses a challenging and, at the same time, an interesting research area. Not only is this a current trend, but also is it extremely beneficial for both the scalability and performance of the framework. This work is also an opportunity to further improve the scalability by making use of more machines for computation and therefore enabling processing of enormous graph data sets. Also, it is unquestionably desirable to maintain the efficiency and performance that was shown in the shared-memory implementation [Stutz et al., 2010]. Hence, the importance of this work is to try to push it to the limit of computation in a distributed way. Therefore, Distributed Signal/Collect tries to add the horizontal scalability that the framework does not currently have.

The goal of this thesis is the design, implementation and evaluation of the Distributed Signal/-Collect prototype which has the ability to benefit from the usage of more machines for the processing of computation and also the usage of larger data sets. Furthermore, this thesis shows and compares the performance of Distributed Signal/Collect evaluated against the shared-memory implementation.

1.2 Outline

This thesis starts with an introduction into the related work in the fields of research for parallel graph processing and frameworks for distributed systems in Chapter 2. Subsequently, in Chapter 3, the design for the Distributed Signal/Collect is outlined, whose implementation is described in Chapter 4. Experiments conducted for the evaluation and a discussion of the results are presented in Chapter 5 and 6 respectively. Finally, Chapter 7 concludes this thesis with a summary of the work done and an analysis of the latter deriving final conclusions and giving future prospects for work to be done.

2

Related Work

In this chapter, an introduction to the related work in the fields of parallel graph processing and distributed systems is given. Different existing frameworks for graph processing are presented in 2.1. Approaches and concepts for making a system distributed are shown in 2.2. Finally, in 2.3 a comparison is made pointing out the challenges that had to be overcome when evaluating the risks and trade-offs between distributed frameworks.

2.1 Existing distributed parallel graph processing

Existing paradigms for parallel graph processing are scarce and each of them tries to approach different specific problems. For example, some of the earlier designs offer a simplistic approach to solve problems with parallel decomposed tasks [Dean and Ghemawat, 2008], even though not all types of problems can be solved this way. Other types of problems, outside of the main semantic web scope, can also be designed to work with graph-like processing with [Low et al., 2010] and [Haller and Miller, 2011] being examples of specific ways for problem solving.

From the existing framework choices, attention is given to the fact that the known frameworks are well featured, offering an acceptable number of properties such as the capability of distributed execution and an asynchronous parallel model for performing various tasks. Thus, the criteria for selecting these frameworks for comparison with our desired solution are given. These in turn, attempt to summarize ideas and point out relevant mistakes such that a correct implementation could turn Distributed Signal/Collect into a viable model for the future.

MapReduce is a recent framework for dealing with large data in clustered and grid tiers. Its simple programming model, however low-level, can be roughly described by only developing the map and reduce functions, where map can be seen as data input and the reduce as the input processing and output of the results. The infamous word count example [1] in MapReduce helps understanding this concept. This simplistic approach, even though used by many, is a limiting one-algorithm-model for all types of problems that often times can be cumbersome to tackle. However, due to its success, it is a notable mention.

Google Pregel is an approach to solve problems involving large graphs. It has a similar processing model to Signal/Collect with some nuances. The basic unit of processing is a vertex and not a worker entity managing many vertexes as in Signal/Collect. Another difference is the usage of sequentially well-defined steps called 'supersteps', making it not a real asynchronous message passing between machines during execution. More details are seen in the next sub chapters. Also, a master coordinator entity exists for computation management, results aggregation and statistics collection.

GraphLab is a new approach into the field of parallel computation. The reason for including it in our related work was the fact that this framework is a different proposition towards parallel computation that takes advantage of graph-like structures for solving problems, specifically Machine Learning problems, using a consistent abstraction. This abstraction tries to address the philosophy of the one-size-fits-all problem solving by proposing a specific framework that deals with graph-based data model. For this reason, it presents a set of characteristics that are worth mentioning in our study.

We continue our detailed study in the subsequent chapters when we try to analyze existing features for creating a bridge between existing works and our proposed solution.

2.1.1 MapReduce

As stated in the introductory section of this chapter, MapReduce is a programming model for processing large data sets. In general, what users need to do is to write a map function and a reduce function. The MapReduce library will automatically parallelize the programs and execute

[1] http://hadoop.apache.org/common/docs/current/mapred_tutorial.html

them on the cluster.

The initial parallelization happens at the input step when the framework divides (or splits) the input into parts, each assigned to a map function. The map function is responsible to assign each division to a user-defined key/value pair mapping. It will then further generate key/value pairs outputs that are given to reducers. Reducers can be regarded as the basic unit responsible for applying the reduce function for each unique key in the values that it contains. The final step is then to output the results when the computation is done, with an optional merging step in between.

A MapReduce instance could be abstracted into a graph. However, this solution can be regarded as a 'shoe-horn' approach. This means the abstraction will work in the framework. The inconvenient step is the modeling of a graph in such map/reduce functions. This works well for data that needs processing in a pipeline style, where data is transformed at each step and going further for more processing based on previous processing. In contrast, such a model does not work well when there are not clearly defined stages for processing data. Therefore, the one-size-fits-all approach has some disadvantages at this point, requiring better alternatives for parallel task-oriented models.

Even though MapReduce imposes this limitation to graph models, the fact that it is a performing distributed system can be regarded as an example with successful use cases as demonstrated in [Chu et al., 2007], [Cohen, 2009] and [Kang et al., 2009].

2.1.2 Google Pregel

Pregel is usually thought of as a parallel graph transformation framework. The most basic unit is a node [2] containing the properties, outward arcs [3] plus the node identifier the outward arc is pointing to. Since the node is the basic unit, it takes the advantage of using a mailbox for receiving messages from incoming arcs. The algorithm starts with the partitioning of the graph. Whenever a graph is loaded into the framework all its nodes are grouped into partitions. Each partition is considered a unit of execution that makes use of a thread for computation.

In contrast, Signal/Collect uses the same model, having the vertexes distributed to worker-

[2]Node being a synonym for vertex in a graph
[3]Arc being a synonym for edge in a graph

threads i.e., the worker uses a thread for computing the vertexes it manages. Therefore, a partition in Pregel can be thought of as a worker in Signal/Collect.

In Pregel's context a machine 'worker' can host many partitions, each having a thread for itself. That way, multiple processing units are executing parallel tasks abstracted within a so called superstep concept. This concept is part of the Bulk Synchronous Processing (BSP) model. For each superstep, the processing unit (partition) receives its messages delivered from the previous superstep such that they can do their computational processing. From this, they can also generate messages that need to be sent to other partitions, all happening in parallel and asynchronously. Whenever messages are delivered, upon execution of the next superstep the execution units can process them. Finally, the synchronicity on BSP comes in the form that supersteps can only start once all processing units have finished delivering their messages. From this point, the cycle repeats until a termination condition has been reached.

Google Pregel uses a concept for finding nodes within worker machines with the aid of a simple hash. This is a similarity to Signal/Collect. In addition, there is a coordinator playing a central role in the execution of the framework which is also an affinity. This is also a necessity for controlling when supersteps should happen, since all machines need to be finished for issuing a new superstep. Other features are also available such as fault tolerance via checkpoint coordination and worker health check heartbeats for reassignment of partitions upon non-response of dead machines.

In many ways, Pregel does indeed have very useful properties desirable in a distributed system. Although it does not offer total asynchronicity in the message passing model, the ability to scale to many machines and its fault tolerancy features puts it into a position that one has much to learn from.

2.1.3 GraphLab

GraphLab is a proposal that deals with a specific problem: solving machine learning algorithms in a parallel fashion. The idea came from the fact that, when existing parallel frameworks are applied for solving machine learning algorithms they somewhat limit the efficiency of the execution [Low et al., 2010]. The solution proposed was a high-level abstraction in a data-graph format that tried to solve parallel algorithms involving a sparse computational structure, a fundamental characteristic of some machine learning algorithms.

Sparseness and parallelism are somewhat tied together. Since data partitions can be stored in the vertexes and the edges of the graph, an association of data can be defined such that the graph representation achieves a high degree of sparseness. In fact, it depends more on the data and how one makes this association, such that the benefit of parallelism can be achieved.

In addition, the functions that the framework provides run concurrently with one another. The sync mechanism, analogous to the reduce operation from MapReduce and the update mechanism, corresponding to the map operation in MapReduce, with some additions, are the user defined core functions.

Another feature that is worth mentioning is the scheduling. For executing the algorithms one needs to define a scheduling plan dictating a desired sequence for execution i.e. the application of sync and update functions. Depending on how the execution steps are established, there could be asynchronous gains in parallelism and also prioritized scheduling mechanisms, all provided by the framework since choosing the right scheduling could be difficult, according to the authors. Therefore, with such techniques in place, the framework takes advantage of parallel executions.

Last but not the least, algorithm termination can be assessed either by non-availability of tasks or user-defined convergence functions that analyze the state of the graph. This is a feature found in almost all graph frameworks discussed so far including Signal/Collect's corresponding termination detection.

2.1.4 Summary

Taking the frameworks listed before, we can clearly see some influences upon Signal/Collect's inner workings implementation. This can be attributed to the way graph-like parallel computations are designed in order to be performing and efficient. The properties available include the parallelism for performing various computational tasks in a graph simultaneously, asynchronicity between units of computation and also a coordinator entity responsible for managing the execution. These features are all included in Signal/Collect and thus we must direct our focus on missing features.

As it is the purpose of this thesis, adding the possibility for distributing the framework and scale horizontally can make Signal/Collect a complete framework, for at least taking it to the same level as existing ones.

2.2 Frameworks for distributed systems

Initial choices of distributed frameworks were mainly directed towards a strategy that would favor a non-central approach when implementing the distributed infrastructure for Signal/Collect. Nevertheless, availability of truly non-central designs was little in the Java/Scala sphere.

Many employable tools considered reputable examples by the community were left out or, at least, were seen as a risk for the project. This was mainly as a consequence of the fact that our approach was more inclined to being a more homogeneous one, such that integration had to be minimal and uncertainties needed to be avoided.

For instance, ZeroMQ [iMatix, 2011] is an interesting solution for queue management system [4] with the philosophy of being simple but at the same time powerful enough for building complex distributed systems. Nonetheless, since the whole library is built on top of C++, even though Java native bindings are available, we tried to avoid risks when implementing a large and complex system over many machines. Also, by keeping Signal/Collect with as few dependencies as possible and, not forgetting, as little integration as possible, we gain portability and plenty of flexibility.

Another consideration was the usage of Message Passing Interface (MPI). MPI [MPI, 2009], [Burns et al., 1994] has been used extensively in parallel computing for many years mostly because of efficiency. Nevertheless, MPI is not suitable for programming in a heterogeneous environment since its underlying structure is optimized for each system and network, making it non-interoperable with different architectures. An attempt to build such an API in Java was made, sadly not a complete one, called MPJ (MPI for Java) [Bornemann et al., 2005], as part of the Java Grande forum providing high performance with the ability for being deployed in multiple platforms.

Having such a scenario, we present a few solutions that were seen as good candidates for the

[4]http://www.zeromq.org/area:results

implementation of this thesis. A comparative analysis follows afterwards where at the same time we formulate arguments for our framework of choice as the base for distributing Signal/Collect.

2.2.1 OpenTerracotta Cluster

OpenTerracotta [Terracotta, 2011] is an open source solution for clustering and replication of Java objects. It allows distribution of objects using an implementation within the Java Virtual Machine (JVM) level. For making an application distributed, Terracotta's server creates a large JVM that is composed of all of the clients' JVMs. This helps the application to replicate changes in objects that reside in another client within the cluster with minimal effort since the job is mostly done by the server controlling the JVM infrastructure.

The approach used to control the sending of data between machines is bytecode instrumentation (BCI). The developer then dictates which parts of the code should be used as shared state such that the server instruments the code only in those parts necessary for the proper distribution of state change. Having said that, the transactions between the JVMs become more similar to the transactions used in the Java memory model. In addition, these transactions contain only the data of the fields that have changed and they are routed through the server to the other clustered JVMs, maintaining consistency. Moreover, the server only sends the transaction to the other JVMs that have objects instantiated on the heap that are represented in the transaction.

Terracotta's strategy is to bypass Java serialization, identifying only the components that change inside the application's objects. In that sense, it saves time in the transaction operation since serialization might be expensive for certain types of data, e.g. large contents with small changes requiring re-serialization of the whole data. Even though this solution makes the replication of only the fields that changed, the usage of this mechanism requires that all data pass through a central server rather than sending data directly to the replicated instances in the cluster.

2.2.2 HornetQ

HornetQ [JBoss, 2011a] is an open source framework providing a solution for clustered and asynchronous message system. Built entirely with Java, it's design is basically based on two messaging principles:

- the message queue principle (also known as point-to-point messaging) and
- the producer-consumer concept.

The message queue pattern is described by having a message being sent to a queue. The queue is tied to a set of clients that can process the messages available in the queue. When using this method, the message can be processed by only one consumer from this set, removing it from the queue when processing is done. This concept takes advantage of the loose coupling between senders (the part who puts the message in the queue) and the receiver (the part who removes the message from the queue).

On the other hand, the producer-consumer pattern has many producers sending messages to a topic on the server. Consumers in turn subscribe to a topic and each subscription receives a copy of each message that was sent to the topic. The difference when compared to the message queue pattern is that a message can now be processed by more than one consumer.

In HornetQ, both concepts guarantee delivery to the consumers and they can also be durable assuring persistence in the event of failure or restart. Also, the design of both patterns is similar, where message queues and topics reside in the server and processing units (clients) for the queues are separate from the server. This guarantees the decoupling between server and clients and ensures messages are not processed twice in the message queue pattern and replication in the producer-consumer pattern.

The default core API and transport layer of HornetQ are the Java Messaging System (JMS) and JBoss Netty respectively. JMS has been a popular API for messaging that provides a standard for a distributed communication, defining message interfaces that attempt to minimize the set of concepts that a developer needs to know. Similarly, Netty defines the wire protocol for the messaging.

The JBoss Netty [JBoss, 2011b] project has gained attention by the fact that it provides an alternative to standard Java Network IO, simplifying development of sockets and offering high performance and high scalability.

2.2.3 Akka

Akka [Typesafe, 2011] is a framework for providing concurrent and scalable architectures on top of the Java Virtual Machine. It is built entirely with the Scala Language [Odersky and al., 2004], benefiting from a variety of features inherent to the language, for example, functional programming, that other languages such as Java cannot provide. These features in turn, help the developer to build concise concurrent applications without much effort with added flexibility.

The concurrency approach that Akka uses is the actor model [Hewitt et al., 1973]. This model is nothing new, dating back from 1973, first appearing in languages such as Erlang [Armstrong and Virding, 1990], later being incorporated into Scala. actors can be seen as the fundamental part of this concurrent programming paradigm. This paradigm is an attempt to relieve the developer from working with locking mechanisms and thread management and also the cumbersome sharing of state. It also helps with the separation of concerns between job processing modules and job providing modules. In other words, this independence is much more clear. For instance, an actor is an entity that receives messages and processes them and, most of the time, they will execute a computational task based on that message. This design is highly useful when thinking about certain development patterns where one have producers and consumers of tasks. The actor model also allows the addition of behavior to an actor where it can also be a producer of tasks, sending messages to other actors.

Although Scala has an actor library, Akka has their own implementation with plenty of features already available for the user such as load and timeout management and also failure handling. Another feature worth mentioning is the asynchronicity of Akka actors, enabling a programming model that is non-blocking and suitably efficient. This is valid for both the local concurrency (vertical scalability) and the remote concurrency (horizontal scalability).

For sending messages between remote actors, Akka uses the already mentioned JBoss Netty, which provides a performing network transport layer over other Java asynchronous network IO frameworks [JBoss, 2011c]. What is important to emphasize is that Akka provides a transparent way of dealing with both remote and local actors, where the framework does the network management behind the curtains but, at the same time, providing all features from message passing to error treatment.

2.2.4 Comparison and summary

From the frameworks selected for comparison, we start by analysing the advantages and disadvantages of OpenTerracotta. Terracotta devises an asynchronous model mainly for Inter-JVM communication which is clearly not focused on data streaming, since it needs to perform byte code instrumentation almost all the time. However simple architecture with promised no code change, the framework uses a central approach where every message needs to be routed through a server that requires fine tuning and complex configuration parameters setting. Mainly for these facts, Terracotta is efficient at keeping object data coherent and available when small and rather infrequent messages need to be transmitted between JVMs [5]. In addition, this shows clearly that this design is not the ideal one for a point-to-point communication system.

Coming now to HornetQ, one of the reasons for including it in our tools comparison and evaluation was the number of features that it includes in its implementation. Asynchronicity on the messaging protocol, the usage of Netty in the underlying transport layer and many other properties made it a serious candidate for serving Signal/Collect with a distributed infrastructure. Unfortunately, HornetQ employs a central server architecture for message passing among clients and, on top of that, client message queues are kept on the server side. This helps with reference uncoupling but, on the other hand, this is a bottleneck that could pose a poor horizontal scalability. Referring back to the producer/consumer model, HornetQ scalability properties can only be seen on the consumer side, where many consumers can be included without much effort. However, our desired design requires that consumers should also be producers. This in turn makes it cumbersome to manage and also puts an overload on the server with plenty of queues to manage at a given point in time.

Last but not the least, we outline the details about Akka. When comparing Akka with other platforms, the first thing that can be noticed is the facility of the framework being simply included as library for accessing its functionalities. Moreover, this lightweightness imposes no effort for integrating it into the Signal/Collect code added to the fact that its Scala nature provides a standard API with all features of the language readily available. Another advantage that needs consideration is that Akka requires no central infrastructure, functioning as a point-to-point protocol when actors need communication between themselves. In spite of that, Akka opposes to the simpler

[5]http://www.terracotta.org/confluence/display/docs/How+DSO+Clustering+Works

management philosophy by not using a single point of management as the infrastructure base. That means that the framework is dependent on a distributed management making the overall infrastructure tightly coupled. Nevertheless, no bottleneck over simplified management is not an unsafe choice for one to make when considering performance.

Table 2.1 summarizes the tools evaluated with some of their characteristics. It gives an overview of the points cited in this sub chapter and it also expresses our needs for what was considered most important when choosing a framework to work with. As can be inferred from this table, all chosen tools deliver asynchronous messaging. However, we are safe to assume that Akka's latency in message exchange is low since it does not devise an architecture with central message routing. Programming effort is considered a subjective matter and thus needs to be taken with a grain of salt. Lock-based concurrent programming and multi-threaded synchronization becomes rather difficult to manage and excessively complex to administer in larger systems. Thus, when the actor model proposes a concurrency philosophy with lock-free mechanisms and also immutable data structures, then it becomes much more inviting and pleasant to develop systems without much strain.

	Terracotta	**HornetQ**	**Akka**
Asynchronous	Yes	Yes	Yes
Basis for Communication	Improved Java Sockets	Netty Sockets	Netty Sockets
Central Server message routing	Yes	Yes	No
Reference Coupling	No	No	Yes
Latency	High	High	Low
Message Throughput	Bottleneck	Bottleneck	Point-to-point
Management	Central	Central	Distributed
Programming effort	High (Java concurrency)	Medium (Producer/Consumer)	Low (actor Model)

Table 2.1: Framework comparison summary

Our investigation concludes that the previously mentioned findings, along with a preference for maintaining overall efficiency of Signal/Collect leans towards choosing Akka over the other candidates. In our understanding, it is inconceivable to take an approach with a central server bottleneck, even though loose coupling is the desirable way to work with, in order to avoid the so called ripple effect [Black, 2001]. Management is also a disadvantage that comes with this choice. However, it is the author's opinion that this is a small price to pay for gaining performance and efficiency when deploying a distributed system. Additionally, Akka has numerous properties that help ease development and implementation of modules for achieving an effective distributed system.

3 Design

The framework choice for this work was the usage of the Akka framework, mainly as a result of the previously mentioned advantages listed. This chapter addresses an overall view of the design choices that were intrinsically related to the framework of choice, and also other considerations related to the distribution of Signal/Collect such as the addition of new features.

A note to the careful reader. From now on, a reference to an actor will be directly analogous to the Akka actor to avoid repetition. Unless explicitly stated, an actor is always an Akka actor throughout the remainder of this work.

3.1 Signal/Collect without distribution

Signal/Collect's initial proposition and evaluation showed remarkable capabilities in terms of efficiency and scalability [Stutz et al., 2010]. This and other characteristics were the main reasons for delving into such an exciting project. Nevertheless, Signal/Collect only proposed a local execution approach, being limited by the number of cores and memory available at a given machine, which becomes impracticable at some point in time demanding expensive solutions for solving a problem.

Consequently, the need for a more capable and powerful model was a priority. No horizontal scalability poses a limitation at how much data can be loaded and also the length of the computational time required. Even though Signal/Collect offers an exceptional, close-to-linear scalability with the addition of more workers for solving a computational task, not being able to load enormous web graphs is a fact that must be addressed properly and is, as stated before, one of the goals of this thesis.

In this sense, what Distributed Signal/Collect tries to accomplish is solving the problem of large scale computational demand. By taking advantage of using resources among different machines we try to outperform local scalability. The remainder of this chapter shows how that could be achieved.

3.2 The actor model and the Akka framework

Not surprisingly, Signal/Collect's design fits admirably well with the actor model where the workers are the basic entity for computational processing. In this sense, adaptations in the framework were minimal with respect to replacing the existing implementation for an actor entity.

In contrast with the original implementation, most of the messaging part, such as inbox processing and queue management were abstracted away since an actor already provides an underlying infrastructure for them. Also, blocking operations such as having to wait for a message to arrive at a worker have disappeared. This is a characteristic pertaining to the actor model, where messages are asynchronously sent using a 'fire-and-forget' philosophy. Naturally there are other message sending mechanisms, for example, the usage of Futures, only they are abstracted within the framework and are indeed used as part of the implementation.

One of the biggest advantages of the actor implementation for the distributed case is the possibility to work transparently with remote actors as if they were local. The mechanism that Akka implements for this is called *Actor Referencing*(ActorRef). The ActorRef entity is the base hook for sending messages and interacting with an actor, so one can work in an unambiguous way with them, whether they are local or remote. The only detail that changes is the way to get this hook (a local or a remote one). This is illustrated by the example codes following next.

For getting the reference of a local actor, the following code is used:

```
val actorReference: ActorRef = Actor.actorOf[MyActor]
```

MyActor is a class that extends the Actor entity and, as one can see, instantiation happens in a special way. What exactly happens behind this code statement is the actor preparation for receiving messages, followed by the start call on the actor reference:

```
actorReference.start()
```

Whenever those statements are executed, message sending is executed via the 'bang' (!) operator, depicted in the code below. The 'bang' operator [Armstrong and Virding, 1990], in Akka has the 'fire-and-forget' semantics which means that no reply is expected:

```
actorReference ! "Hello"
```

One design for creating messages to be used between actors exists as a common programming practice. Messages can be easily coded as case classes in Scala. Case classes export their constructor parameters providing a recursive decomposition mechanism via pattern matching. This in turn helps for message matching at the actor side with the use of the receive function. [1].

For the remote actor, the machine containing the actor instance, i.e. the one providing the actor service can be seen as the server, and the machine using the remote instance can be seen as the client, not being limited to have server and client in separate machines - i.e. remote actors can also be accessed in a local way.

To instantiate an actor on the server side, we also make use of the special *actorOf* statement. In addition, the usage is closely related to a socket where the IP address or a hostname plus the port where the actors will listen are the only parameters required (for the most simplistic case). When registering remote actors, they are automatically started (the usage of the start function is no longer needed):

On the server machine:

```
remote.start("hostname", port)
remote.register("service-name", "hostname", actorOf[MyActor])
```

Since there can be more than one actor being available remotely and, most importantly, to disencumber the client from knowing which class is being used by a remote actor, the usage of a service name as a parameter greatly simplifies the configuration for getting the correct remote ActorRef. It can also be used as an intuitive way of getting the behavior of the actor. To instantiate

[1]http://www.scala-lang.org/node/107

more actors, all one needs to do is call the register function (the start function only starts the server). Access to the remote actor happens via the code below.

On the client machine:

```
val ref: ActorRef = remote.actorFor("service-name", "hostname", port)
```

In addition to all the foregoing, sending messages could be done via the same ’bang’ operator. But now, since we are dealing with networked actors, mechanisms should be put in place to ensure correctness in the event of network failure. The ’bangbang’ operator (!!) semantics is used to send a message asynchronously to the remote actor but it will wait until a reply is received or a timeout is reached.

3.3 Distributed Signal/Collect architecture overview

Having explained how actors can be instantiated and how the actor model works with message sending, we present the Distributed Signal/Collect architecture. Mainly, the core architecture, when related to algorithm execution and core functionalities, have had no drastic changes. One can consider an almost identical architecture in the distributed case, taking into consideration only those parts necessary for the correct network deployment and distribution.

To start the infrastructure for the compute graph execution in a distributed way, all machines start simultaneously executing a bootstrap sequence that comprises of machine discovery and a simple leader election algorithm. Once the leader is selected, the other machines become ’zombies’, waiting for the leader’s ’orders’. For sending ’orders’ to the zombies and receiving zombie ’acceptance’ of the ’orders’ or ’requests’, we use actors as machine managers, which are responsible for managing the correct initial startup and instantiation of remote workers. Once workers are initiated, the leader will know how to get their references via the bootstrap mechanism that has been currently undergoing.

After proper initialization, the managers are kept running but only in order to be a way of coordinating correct shutdown of the infrastructure whether it is due to a failure or due to proper termination. The leader will also contain all the coordinator logic since Signal/Collect depends on a coordinator for algorithm correctness and consistency. For instance, the zombie machines communicate with the coordinator residing at the leader via a remote actor which will simply forward all messages to the real coordinator implementation. These forwards are, namely, worker status and algorithm control messages (e.g. start, pause and stop).

Figure 3.1 presents a general overview of the infrastructure that is available at the beginning of the execution. These distributed management entities, particularly the managers, are mostly used at the beginning of the execution which is in fact happening at the bootstrapping phase.

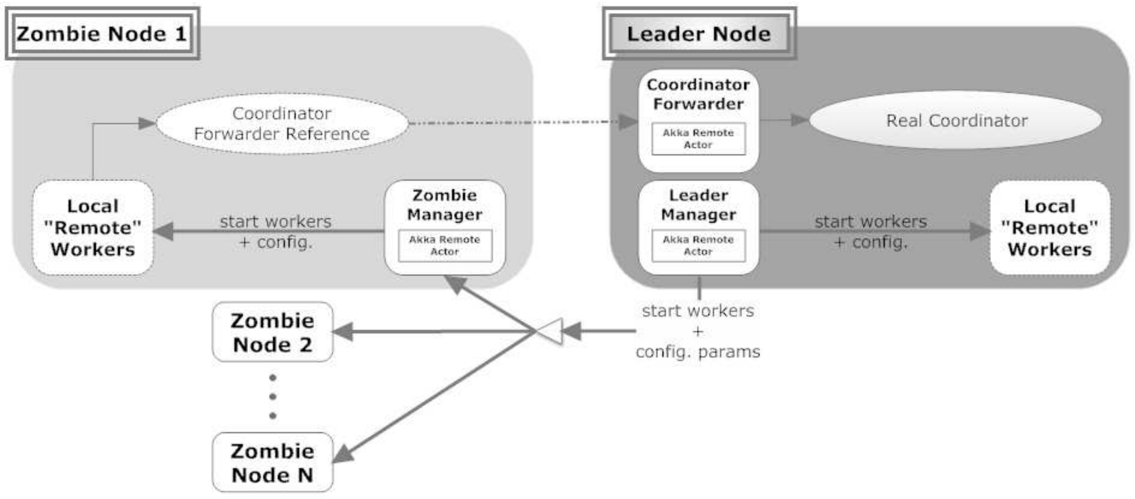


Figure 3.1: Architecture overview - Distributed Signal/Collect

In the next chapter, a more detailed explanation of the mentioned bootstrap mechanism is given, as well as other message exchange phases important for the correct infrastructure activation.

4 Implementation

This chapter shows how the proposed design of Signal/Collect with distributed mechanisms and the actor model was implemented, taking the improvements described in the previous chapter as reference. An overview is given of the most important Distributed Signal/Collect components, based on which it is shown what was added to achieve a running version of the distributed system that realizes the design described in the previous chapter.

4.1 Modifications in Signal/Collect for distributed support

The class diagram in Figure 4.1 depicts a general overview of the classes that are core extensions related to the distributed execution of Signal/Collect. It also presents a rough overview of the newly implemented classes that provide feature additions related to the distributed mechanisms. Although the class diagram is incomplete, it gives a good overview of the most relevant features and properties, which were added as part of the distributed architecture. Specific implementation of the related classes is shown in the following sections.

4.1.1 Akka Worker

The worker class has been extended to meet the requirements imposed by Akka's actor model. To a degree, the actor has a receive method that processes a message from its mailbox. This method gets executed whenever a message is available for processing. From an asynchronous point of

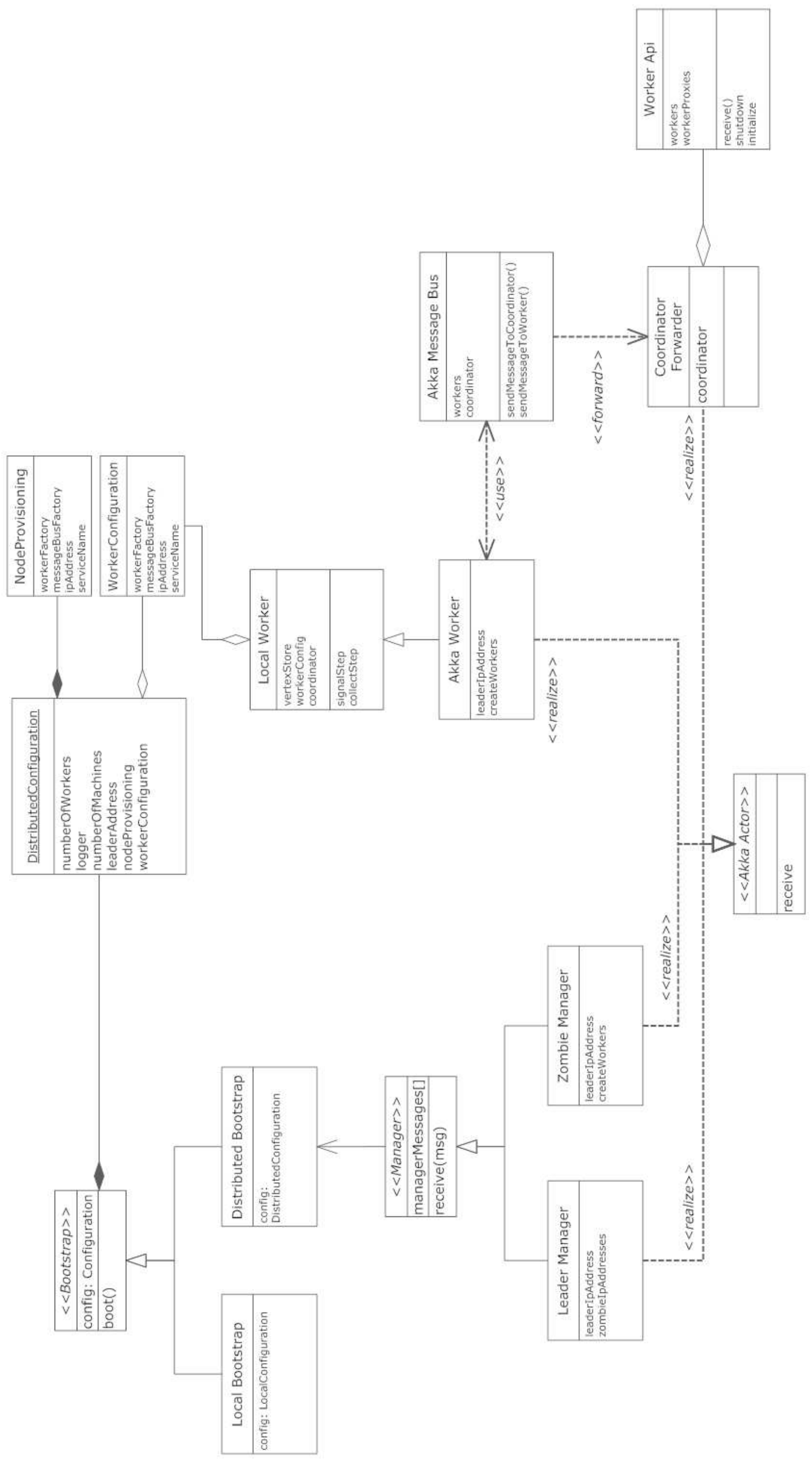


Figure 4.1: Simplified Class Diagram

view, this is the perfect solution for sending messages to an entity since they get executed at the actor in a FIFO manner, as long as they are in the actor's mailbox.

On the other hand, this changes the semantics of the original worker implementation. In the original worker, algorithm execution has to be coordinated with mailbox check, message processing and the computation step (signal and collect steps). A separation between mailbox management and message processing is not clearly identifiable at first hand. However, the gain in performance was proven to be high since messages are processed as long as they arrive in the mailbox, giving the worker a good throughput rate at processing [Stutz et al., 2010].

Returning our focus to the receive method, a mechanism had to be implemented to escape the computation step or, in other words, the core computation functions of Signal/Collect. For instance, a mailbox check is made [1]. Upon emptiness, Signal/Collect execution continues normally. Whenever the mailbox contains a message, execution is halted to give space for the receive method to be called again (so that a message gets taken from the mailbox and processed). Above all, it is important to emphasize that by implementing such procedures we are guaranteeing consistency and similarity with the original implementation.

Going further with changes and features added to the worker, we come to idling detection. The actor library has a feature called *ReceiveTimeout*. Its behavior is a simple message being sent to the actor whenever a predefined timeout is reached. The timeout countdown is triggered whenever no more messages are available at the mailbox. Therefore, if no messages are available to be processed and no more computations need (signals and collects) to be performed, the worker is considered idle.

Last but not least, what is important to remember about this worker implementation is that it can be devised either locally or remotely. No further special functions or parameters are required to be implemented at the worker side in order to be compatible with different architectures. Hence, the addition of the Akka worker gives the possibility for having another worker design for working locally.

[1] A contribution by the author was made to the Akka open source for inclusion of such a feature

4.1.2 Worker Proxy

Changes in the worker proxy were specifically for treating a special case of remotely located workers. Upon coordinator message sending, a reply is sent from the workers to the worker proxy in order to acknowledge receive of the message, thus releasing the proxy from the blocking mechanism that waits for this reply. On dealing with remote workers, messages can be lost or a machine might have crashed. So, a simple message reply await timeout was implemented for the framework to notice the failure and, later on, when fault tolerant mechanisms are in place, deal with such failures.

4.1.3 Akka Message Bus

The message bus had to be replaced by an Akka implementation since message sending is not done via the *MessageRecipient* entity. That is obvious since now we use the 'bang' operator for sending messages to the workers. In addition, for correct communication between remote actors and the coordinator, the reference to the coordinator forwarder is used instead as a message passing to the real coordinator staying at the leader.

Message serialization is also implemented to use standard Java as part of the Akka library. Some changes in the way the framework sends messages, mainly due to closures offered by the Scala language, had to be made in order to avoid serializing unnecessary data and also to increase performance.

4.1.4 Bootstrapping

A bootstrapping mechanism has been made available for correct startup of the infrastructure necessary for the computation. That is an abstraction for the user to only worry about choosing the desired architecture, either local or distributed, such that the mechanism takes care of the correct steps for initialization.

For clarification, the required steps for correct bootstrap are presented. Upon declaration of configuration parameters for execution (described in the Appendix A.1), the first and most important one is an initial setup for worker communication when there are no means for getting their references by usage of Remote Procedure Call (RPC) and Worker Proxies. [2]. Since this process needs an exact sequence of steps, this is hidden from the user so that errors are avoided at the most elementary steps. Subsequently, the coordinator is created and all the workers can get the references to the other workers for message passing, which is also a meaningful step. It is important to remember at this point that the coordinator is only instantiated at the leader machine, which in turn is similar to the shared memory execution (only one instance should exist). Finally, the compute graph is created and returned for direct manipulation. The compute graph is available as an API where the user decides to add vertexes and edges for the graph computation according to an algorithm of choice. More details can be seen at the original Signal/Collect paper [Stutz et al., 2010]. Examples of usage can also be found in the source code under the examples package.

4.2 Implementation of distributed mechanisms

4.2.1 Machine discovery

At the bootstrap phase there is an implementation that addresses the problem of computation machines finding each other. This is the case since we are using a job scheduling mechanism at our test bed and the jobs sent to it can be executed at different intervals and assigned to different machines. Hazelcast [3] is a library for easily setting up clustered data distribution among a series of machines. Initial tests demonstrated that Hazelcast does not offer a very high performance for fully implementing a distributed system, although it offers discovery using multicast or TCP/IP. The fact that it is not performing for the whole solution does not impose a limitation for the bootstrapping since such step does not require high performance. According to our measurements, it takes a maximum of 30 seconds for 12 machines to find each other. For this reason and also for the fact that it supports commercial tools for cloud computing such as Amazon EC2, we decided to include it in our solution.

The initial step is the creation of a Hazelcast cluster that finds running instances in other ma-

[2]This design has been recently implemented in Signal/Collect
[3]www.hazelcast.com

chines. Consequently, each machine has the knowledge of all the cluster members to send them a randomly generated identifier. A bootstrap manager is responsible for the send and also reception of the pair [*IP Address* $\rightarrow$ *Identifier*] respectively to and from all machines.

Upon receiving all pairs, each machine compares its own identifier with those received and the IP address with the smallest identifier is selected as the leader. The bootstrap sequence continues upon deciding if the machine should work as a zombie or as a leader, discarding the bootstrap manager since its task has been fulfilled.

4.2.2 Machine management

The machine management module is used only at the worker creation step, distributed graph loading and termination of running instances. All the modules were implemented using actors: the leader manager and the zombie manager.

The leader manager, existing at the leader selected machine, takes care of sending all the zombies the configuration parameters for the creation of workers. It also functions as a checkpoint entity for coordination of zombies, namely to check if the zombie managers have come online and if the remote worker creation was successful.

What the zombie manager does then, is to receive the proper configuration from the leader, instantiate all designated workers and rendezvous with the leader saying it is ready for the computation to begin.

These manager entities are kept alive until the end of the computation execution such that termination occurs without problems. Also, they can be later extended for graceful shutdown of remote workers and also serve as a communication point for relocation of resources.

4.2.3 Machine provisioning

Opening space for further extending the cluster distribution of the framework, a machine provisioning method has been introduced. In turn, the initial approach proposes a simple way of distributing workers and vertices among the machines such that a 'fair' provisioning of resources is used. For instance, each machine receives an equal number of workers with the exception of the leader elected machine in case an odd number of workers is used. This was a simplistic design choice and is open to extending by straightforwardly implementing a new provisioning factory that should provide the right details of how to instantiate workers at the machines.

Also inserted in the context of machine provisioning, not only workers can be load balanced, the logic for distributing graph parts could also be added. What matters most is to have an intelligent way of assigning vertexes to workers so that neighboring vertexes could reside at the same machine. This saves time between workers when no messages need to traverse the network [Malewicz et al., 2009] [Xu et al., 2009].

5 Evaluation

This section describes in which environment the experiments were carried out and what was undertaken to achieve the results presented in the following sections. We first describe our test bed, followed by the scenario description and finally with the presentation of the results.

5.1 Test bed

Experiments were conducted in the laboratory of the Dynamic and Distributed Information Systems Group at the University of Zurich. The test bed named Kraken is comprised of 12 clustered machines each having twenty four AMD Opteron(tm) Processor 6174 operating at 2.2GHz plus 64GB of RAM memory. The machines are backed by a Gigabit Ethernet dedicated LAN and could be accessed via ssh or, more preferable, via TORQUE, a cluster resource management system.

The operational system used was a Debian 64Bit with kernel version 2.6.32-5-amd64. For running our Scala programs, a Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode) was available. Heap size was adjusted according to the size of the graph loaded into a given Signal/Collect instance. Also, the option for running in 64Bit mode d64 was given as a JVM argument.

5.2 Test algorithms and used data

Two algorithms were used for validating our findings, such that a comparison could be made to those tests performed at the time of publication of Signal/Collect. The algorithms selected were PageRank [Page et al., 1999] and Single-source Shortest Path (SSSP) [Cherkassky et al., 1994], both well-known graph related problems. The data used for our tests were two randomly generated web graphs with log-normal distributed out-degrees drawn from:

$$e^{(\mu+\sigma N)} \tag{5.1}$$

having PageRank with $\mu = 3.0$ and $\sigma = 1.0$ and SSSP with $\mu = 4.0$ and $\sigma = 1.3$, and N drawn from a standard normal distribution for both. The number of vertices contained in both graphs was 100'000 and the number of edges included was 3'287'988 for PageRank and 12'798'293 for SSSP. Signal/Collect score-guided computation was used with a signal threshold of 0.01 and collect threshold of 0.0.

5.2.1 Early evaluations

In order to determine whether Akka was a feasible solution, replacing the original worker's behavior when using it distributed, or not, an initial Akka Worker versus Original Worker evaluation was made. Early results showed that Akka had great potential to even completely replace Signal/Collect's original worker implementation since it hides a lot of development details and reduces code complexity, helping code maintenance in the long term.

Graph 5.1 shows a comparison of Akka shared memory implementation against the original shared memory implementation in terms of performance, and graph 5.2 shows comparison in terms of scalability.

As it can be inferred from the graphs, almost the same scalability is achieved by the Akka worker. Since Akka offers high scalability as a feature, we were not surprised to see such results. In the performance graph, Akka has also performed well, being slow by an average of 10% in our tests. Comparison in the performance graph should be made with tests that had the same execution mode. Hence the two pairs of lines (for the synchronous and asynchronous).

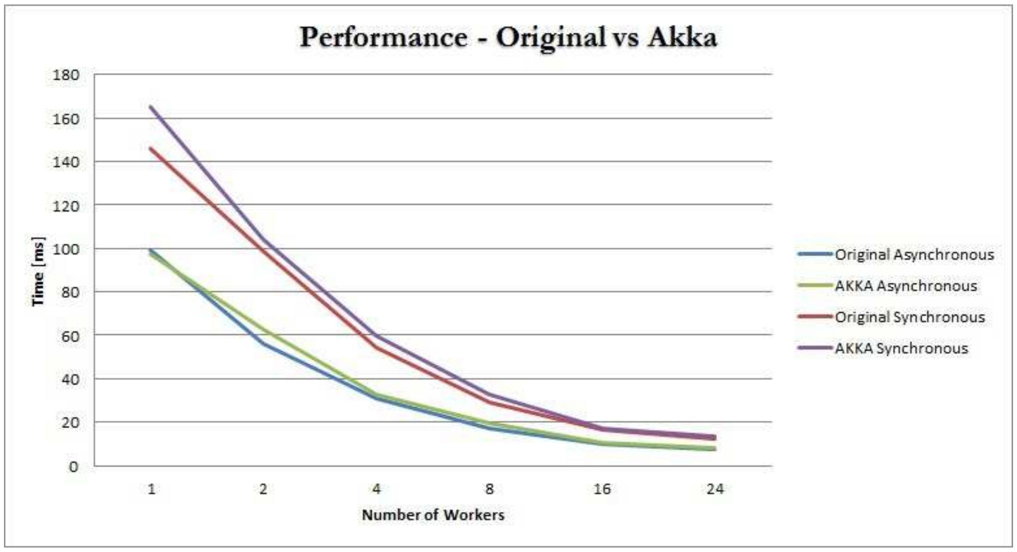


Figure 5.1: Performance comparison - Original vs Akka

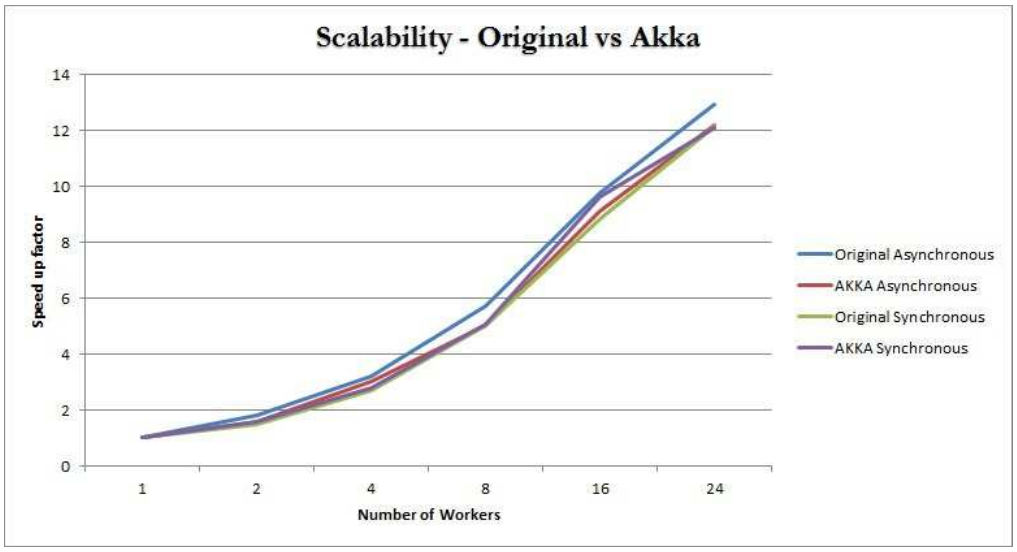


Figure 5.2: Scalability comparison - Original vs Akka

The comparison was made similar to those tests performed at the time of publication of Signal/Collect with an asynchronous and a synchronous execution mode. Since this was a proof of concept, we limited our test scenario to running PageRank only.

5.3 Distributed Experiment Procedure

Experiments were conducted in a sequential manner so that the performance and the scalability of the solution could be evaluated. Since each worker was assigned to a single thread and each thread to one available physical core, we increased each run with a step of 24 workers up to 8 machines, totaling 24 workers for the first experiment and 192 workers for the last experiment. Hazelcast discovery mode was set using TCP/IP by setting in the configuration file all known hostnames, in our case, the 12 machines available at the cluster. The version of Akka used was 1.2-RC3 and configuration parameters can be found in the Appendix A.3.

The asynchronous mode of Signal/Collect was given priority for our test scenarios since it was desirable to test asynchronicity behaviour of the framework over messages going through the network and also taking for granted that asynchronous executions are supposedly more performing than synchronous ones [Koller and Friedman, 2009].

The scenarios, their results and respective interpretation are presented in the following section.

5.3.1 Results

Based on the procedures and the previously mentioned parameters, this section describes the experiments' results and tries to reason them. The results are presented from two different views: performance gain and scalability gain upon addition of machines with more workers. The values shown depict the average value per test over all runs.

As we can see from the graph 5.3, simply adding one machine to the system decreases performance by a factor of almost 100 fold in both PageRank and SSSP. However, further adding more machines to the computation, we can perceive improvement in the time taken for the computation in the same graph size. In PageRank, the loss in performance is more visible than in SSSP, where SSSP almost achieves the same level of performance of one machine with the addition of eight machines.

The speedup or scalability calculation was done by having the baseline as the result of the computation with one machine. Since from graph 5.3 we see a loss in performance, we decided to show the scalability gain without the inclusion of the first computation, that is, without comparing with the shared-memory implementation, to demonstrate that even though we lose perfor-

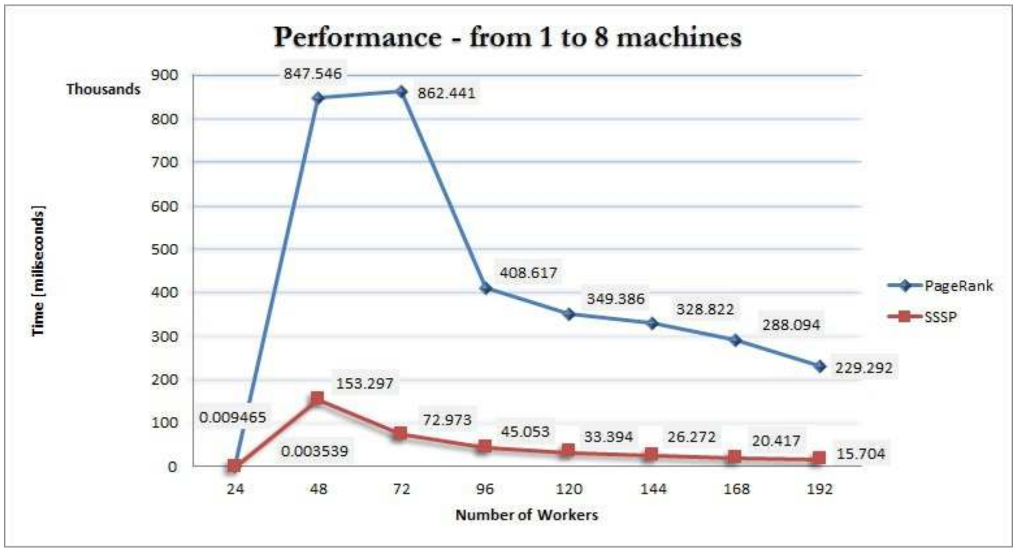


Figure 5.3: Shared-memory against distributed

mance by adding more machines to the system, we can still achieve a certain scalability degree. These results are shown in graph 5.4.

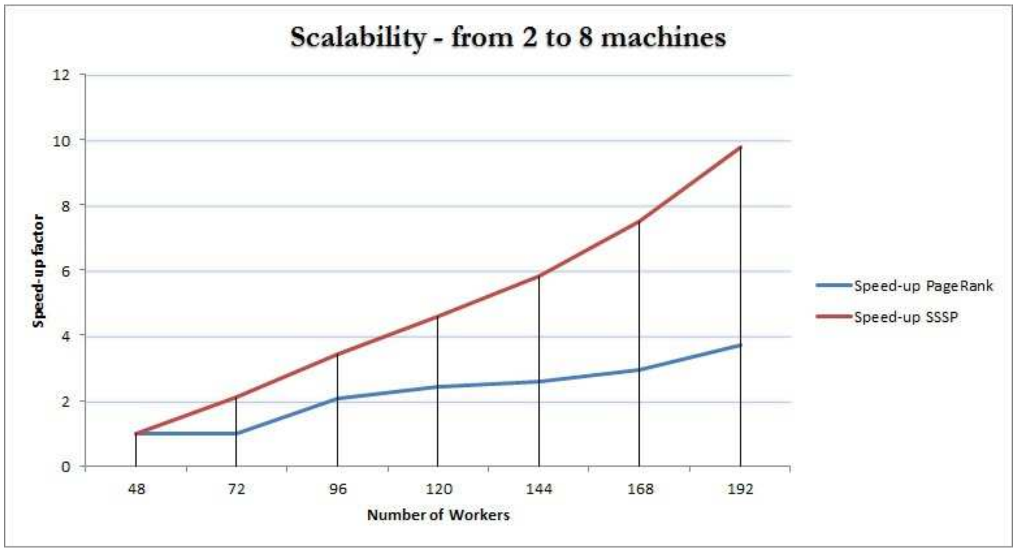


Figure 5.4: Scalability results - Distributed runs only

6 Discussion

6.1 Results

The discoveries from the evaluations show that the underlying framework backing Signal/Collect for implementing distribution of messages have unsatisfactory results in terms of overall performance whenever we added more machines to the computation. This can be attributed to a series of reasons that we will try to describe now.

Network communication is an expensive computational operation, however not up to the point of slowing down computational time to a factor of 100 fold. Akka has shown remarkable initial results at the time of selecting the framework of choice. Such tests demonstrated that Akka is satisfactory enough to sustain a constant network throughput and also to achieve a very low RTT within a dedicated network using multiple machines. Results and code examples can be found at: [de Freitas, 2011].

Another reason could be the use case which was used on top of the framework, namely the Signal/Collect message sending pattern. More in depth knowledge of the Akka framework would be required to analyze such pattern and how it properly behaves when having to cross the network. Another point is to specifically analyze the feasibility of such a framework when dealing with graph computations; although that was not on the scope of this work, some preliminary evaluations were made to determine usage potential as it has been previously stated in 5.2.1.

To be on the fair side, even though we had lower performance, we could achieve a certain degree of scalability when adding more machines. This could be explained by adding more local interactions between neighboring vertexes being allocated to the same machine. Since in-memory message passing is faster than network message passing, we clearly see the performance increase.

Looking from a cost/benefit point of view, such behavior is not desirable, specifically, having to add more machines to be as performing as one. However, this demonstrates potential that the tool has to contribute even more, as long as issues are remediated, enabling it to become the ideal solution for our use case.

What could also be an addition of value is to test viability of loading a huge graph size and distribute it into a high number of machines. The outcome could show interesting results and would be desirable for efficiently checking the distributed solution. Due to time constraints and also lack of proper resources, this test was disregarded.

Distributed Signal/Collect, even though not acceptably performing, adds advantages in terms of features putting it on par with existing frameworks. Comparing with Google Pregel, we determined acceptability of the asynchronous message sending, a feature also present in the framework, although Pregel use a synchronized checkpoint[Malewicz et al., 2009]. In a parallel with MapReduce for example, Distributed Signal/Collect can now distribute the computation to many machines, a feature that has been facilitated by making Signal/Collect distributed.

6.2 Limitations

Some limitations were identified in Distributed Signal/Collect. They either are features still to be implemented in the future or limitations imposed by the architecture design that was created.

Worker message passing is done transparently the same way whether a worker is remotely located or locally instantiated. For this reason, no tight control of worker availability during computation has been imposed i.e., a complete cluster management mechanism was not implemented. Such a feature could in turn ease dealing with failures in the network, machine crashing and rearrangement of cluster resources.

On dealing with failures, we proposed a simple crash handling for our distributed system but no fault tolerance is in place. For this reason, a crash of any machine during the computation fails the execution, inclusive the leader coordinator.

Graph loading is also another issue which is a problem of its own. In order to gain performance, trying to avoid sending messages to the network between machines is desirable. Hence, to minimize time spent on message passing during the computation, having neighboring vertexes staying at the same machine is highly beneficial. To achieve that, a correct graph partitioning mechanism would have to be implemented to allow such an advantage in terms of locality in message passing.

A simple mechanism for distributed graph loading could also be implemented as part of the provisioning. Since a Network File System (NFS) is in place on our test bed, plus the fact that it allows parallel readings from many machines in the network [1], this was considered to be a reasonable solution. The leader delegates which parts of the file each machine has to read and this read happens in parallel at a certain portion or file segment already available from the NFS. When the read is completed, all machines notify the leader that they are ready for starting the computation. Such a feature, the way it is depicted would, in fact, only speed up the graph load time. However, such an implementation combined with graph partitioning would add value to the solution as whole.

Personally, the work that was implemented was already challenging but also pleasant. Evaluating different frameworks, outweighing pros and cons and analyzing risks was indeed satisfactory, to say the least. Even though results were not as expected, I believe that such failures in achieving the desirable effects are just problems that have yet to be solved with future works. Choosing Akka as a framework is also not regrettable since it offers many helpful features and it is under highly active development by a fast growing community. It is under the author's opinion that the community will do a lot within their grasp to come up with a much more sophisticated and performing solution.

[1] http://tools.ietf.org/html/rfc5661

7 Conclusion

This last chapter shortly summarizes the work done and draws final conclusions based on the results achieved in the Evaluation chapter. It also gives a forecast about future work that could be done to improve Distributed Signal/Collect.

7.1 Summary

In order to achieve the goal of this thesis, it was given thought about the necessity of having a transparent system for the user, on regards to whether the user is running Signal/Collect locally or distributed. Existing graph processing systems such as the ones described throughout this work, offered solutions that have as a main goal to process large quantities of data. From that came the motivation to transform and provide the same features for Signal/Collect, while still not having to add complexity, leaving the framework without drastic changes.

Analysis and evaluations were performed prior to the execution of the main implementation to better understand the problem and try to avoid risks inherent to distributing a system. For instance, some frameworks proposed a central model of message routing and also a central server architecture that was mainly avoided due to performance concerns. Also, having the distributed solution as light as possible was a priority such that, as previously stated, a transparent usage could be achieved.

Akka has proven to provide features that helped in development and also gave some basic building grounds for the whole distributed architecture. The Actor-model facilitated development of concurrent and parallel behaviors mainly due to the fact that Akka has all the details of implementation hidden away from the user. Such a piece of middleware is being acclaimed by the community as one of the largest Scala projects to date.

The framework of choice has shown potential to provide a good basis for achieving a good performing system. Nevertheless, our tests concluded that the underlying mechanism still has some weaknesses when applying specific use cases on top of it. Our algorithms for evaluating the solution, for instance, PageRank and SSSP, popular problems in the graph processing area, have demonstrated message passing patterns previously not know to the Akka community. More details and an in-depth analysis of the problem would have to be performed in order to consolidate the framework to solve our specific problem. Newer versions of the framework can also be awaited with wide arms open since they will certainly bring new features and hopefully, more performance. We will discuss them in the future work section.

Last but not the least, some limitations were listed; properties and features that our prototype does not fulfill at this present moment and serve as basis for future works.

7.2 Conclusions

The thesis shows that it was possible to successfully integrate a distributed system on top of an existing shared memory only local solution that can be leveraged to scale up, even though overall performance was not as expected.

Unfortunately, the evaluation procedures could not reproduce real environment data due to the limitation in time since a 100 fold computation time, even in small graphs could take days with the test bed available. For instance, it would be desirable to run the PageRank algorithm on the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph [1] that is available for research purposes. That would have truly allowed us to test feasibility of a large scale system and completeness of the solution with respect to scalability.

Nevertheless, the design and implementation of the distributed extension for Signal/Collect to support message sending through the network and also addition of cluster management modules and bootstrapping mechanisms have also brought up a flexible frame and starting point for the integration of new or other modules that have more tight control over a distributed com-

[1] http://infolab.stanford.edu/ ullman/mining/2009/YahooData.pdf

putation of a graph. Additionally, the usage of an underlying framework to control distributed mechanisms has shown that Signal/Collects architecture has been carefully designed and is open to pluggable modules and also further extensions. Moreover, other distributed frameworks or even a distributed solution that could be done from scratch could also be viable for comparing different solutions with this prototype to determine other potential features not currently available for making Distributed Signal/Collect as much complete as possible. However, it would take time in order to develop a mature and fully featured solution.

Also worth mentioning was the lack of solid frameworks and tools for a qualitative development in the Scala language. Even though the language has developed in the last few years, it was only towards the end of this work that a more powerful compiler and a more stable release of the developer environment were made available to the developers community.

7.3 Future Work

Akka is a work in progress. At the time of this writing, Akka plans to provide Cluster management and better asynchronous message semantics with futures and a new concept called Dataflow concurrency [2]. These features were all part of a previously available commercial only solution offered by the inventors of Akka. With such additions, it is very likely that more and more details will be abstracted away for an even better code maintenance and addition of new features from the middleware to be included in Signal/Collect.

Also, due to the new features made available for the distributed execution, new extensions for the fault tolerance part could be implemented. For instance, the coordinator could detect dead workers via specified intervals and attempt to recover them or declare them unusable. With that, relocation of workers could take place and the computation could continue without too much effort. Similarly, the addition of checkpoint mechanisms would be a valuable feature that would allow rolling back to a certain point into the computation, preserving consistency of the computation in the event of failures. For that, persistency modules would have to be adjusted in order to determine what and how frequent such checkpoints should be done since disk access is expensive, not to undermine overall performance.

[2]http://www.gpars.org/guide/guide/7.%20Dataflow%20Concurrency.html

In case the coordinator machine is the one failing, a leader re-election could take place to decide which node could be the new coordinator/leader. With all these fault tolerant mechanisms we would achieve seamless computation flow to avoid restarts and also to keep the computation running as long as possible.

In addition to the fault tolerance properties, another desirable feature would be to extend provisioning methods. Essentially, new methodologies could be added to better distribute workers to member machines such that physical and logical resources are taken into account. For example, one methodology could take into consideration the number of CPU's available so that more workers can be instantiated. If a machine has a decent amount of main memory, it could be also a way to better load graphs in it and distribute vertexes from this machine.

Besides the already mentioned suggestion for the simple loading the graph using an underlying filesystem, such as NFS, better distributing algorithms can also be implemented. That would also have an impact in the coordination for better partitioning the graph such that the graph load time is not hindered. One possible solution is to keep the data loaded into main memory inside a grid infrastructure that would provide the data on a timely basis or even distribute the load to those machines part of the computation. Clearly, the grid link distribution would have to meet some requirements such that network traffic does not cause delay into distributing the data to the computation cluster.

Last but not least, integrating security measures into the framework would be another valuable improvement. Akka already has some features in place such as the use of secure cookies for message accepting only between trusted actors. However, in the case malicious peers would have access to the network where the computation is being performed, other counter measurements would have to exist in order to secure the infrastructure. Such a scenario is unlikely to happen since execution will be in most cases constrained to a trusted network. However, if it is desirable to achieve large scale and use a public network infrastructure, then it is logical to consider such scenarios.

A

Configuration

A.1 Distributed Signal/Collect configurable parameters

These are the available parameters for Distributed Signal/Collect. They have been modified accordingly to reflect our evaluation procedures.

- numberOfMachines - specify the number of machines to be used for a computation

  from **1** to **8**

- workerFactory - specifies what type of worker should be used

  AkkaRemoteReference - for instantiating remote workers at the zombies

- nodeProvisioning - specify how to allocate resources in the cluster

  EqualNodeProvisioning - for instantiating remote workers at the zombies

A.2 Hazelcast configuration

Hazelcast configuration setting for locating nodes on a network (only those who are intended for use).

```
hazelcast.xml
...
```

```
<tcp-ip enabled="true">
<hostname>claudio01</hostname>
             <hostname>claudio02</hostname>
             <hostname>claudio03</hostname>
             <hostname>claudio04</hostname>
             <hostname>claudio05</hostname>
             <hostname>claudio06</hostname>
             <hostname>claudio07</hostname>
             <hostname>claudio08</hostname>
             <hostname>claudio09</hostname>
             <hostname>claudio10</hostname>
             <hostname>claudio11</hostname>
             <hostname>claudio12</hostname>
</tcp-ip>
...
```

A.3 Akka configuration

Parameters used for running an Akka instance. The configuration file is offered with the distribution file and can be passed as a reference to the program execution via **-Dakka.config**=path/to/akka.conf

```
akka {
  version = "1.2-RC3"
  enabled-modules = ["remote"]
  time-unit = "seconds"
  remote {
    compression-scheme = ""
    server {
      port = 2552
      connection-timeout = 100
    }
    client {
      buffering {
        retry-message-send-on-failure = off
      }
      reconnect-delay = 2
      connection-timeout = 100
      read-timeout = 100
    }
  }
}
```

List of Figures

List of Tables

Bibliography

[Armstrong and Virding, 1990] Armstrong, J. and Virding, S. (1990). Erlang - an experimental telephony programming language. In *Switching Symposium, 1990. XIII International*, volume 3, pages 43 –48.

[Black, 2001] Black, S. (2001). Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 263–279.

[Bornemann et al., 2005] Bornemann, M., van Nieuwpoort, R. V., and Kielmann, T. (2005). MPJ/Ibis: a flexible and efficient message passing platform for Java. In *Proceedings of 12th European PVM/MPI Users' Group Meeting*, pages 217–224, Sorrento, Italy.

[Burns et al., 1994] Burns, G., Daoud, R., and Vaigl, J. (1994). LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386.

[Cherkassky et al., 1994] Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1994). Shortest paths algorithms: Theory and experimental evaluation. In *SODA*, pages 516–525.

[Chu et al., 2007] Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., and Olukotun, K. (2007). Map-reduce for machine learning on multicore. In Schölkopf, B., Platt, J., and Hoffman, T., editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, Cambridge, MA.

[Cohen, 2009] Cohen, J. (2009). Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11:29–41.

[de Freitas, 2011] de Freitas, F. (2011). Akka microbenchmarks by francisco de freitas. https://github.com/chicofranchico/akka-microbenchmarking2.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113.

[Haller and Miller, 2011] Haller, P. and Miller, H. (2011). Parallelizing Machine Learning- Functionally: A Framework and Abstractions for Parallel Graph Processing. In *2nd Annual Scala Workshop*.

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[ILK, 2011] ILK, R. G. (2011). Google Graph Size - Tilburg University, Netherlands - Last visited: 30.07.2011.

[iMatix, 2011] iMatix, C. (2011). Zeromq - http://www.zeromq.org/. High performance solutions - http://www.imatix.com/.

[JBoss, 2011a] JBoss (2011a). Hornetq - an opensource message oriented middleware. http://www.jboss.org/hornetq - The JBoss Community.

[JBoss, 2011b] JBoss (2011b). Netty - an asynchronous event-driven network application framework. http://www.jboss.org/netty - The JBoss Community.

[JBoss, 2011c] JBoss (2011c). Netty performance test reports. http://www.jboss.org/netty/performance - Last visited: 14.04.2011.

[Kang et al., 2009] Kang, U., Tsourakakis, C. E., and Faloutsos, C. (2009). Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238.

[Koller and Friedman, 2009] Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

[Low et al., 2010] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California.

[Malewicz et al., 2009] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2009). Pregel: a system for large-scale graph processing - ”abstract”. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC ’09, pages 6–6, New York, NY, USA. ACM.

[MPI, 2009] MPI, F. (2009). Mpi - message passing interface. The MPI Forum - http://www.mpi-forum.org/docs/docs.html.

[Odersky and al., 2004] Odersky, M. and al. (2004). An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. W. (2010). Signal/Collect: Graph Algorithms for the (Semantic) Web. In Patel-Schneider, P., editor, *ISWC 2010*, volume LNCS 6496, pages pp. 764–780. Springer, Heidelberg.

[Terracotta, 2011] Terracotta, I. (2011). Openterracotta. http://www.terracotta.org/ - Terracotta, Inc.

[Typesafe, 2011] Typesafe, I. (2011). Akka - a platform for writing simpler correct concurrent applications using actors. www.akka.io - Typesafe Inc.

[Xu et al., 2009] Xu, W., Wu, Y., Xue, W., Zhang, W., Yuan, Y., and Zhang, K. (2009). Horde: A parallel programming framework for clusters. In *Web Society, 2009. SWS ’09. 1st IEEE Symposium on*, pages 96 –101.