Distributed Algorithms report

Stefano Varotti, Anis Nouri

June 2015

1 Introduction

The k-core number or the degeneracy of a graph is the largest subgraph in which every vertex is connected to at least k other vertices within the subgraph.

The K core decomposition algorithm serves as a graph analysis tool able to highlight interesting structural properties that are not captured by the degree distribution or other simple topological measures in large scale complex networks. The state of the art algorithm has worst case linear complexity, so when the size of the graph is huge, this sequential algorithm does not scale well and can become unpractical to compute.

This report introduce a distributed algorithm to compute the k-core decomposition of a large size network. In order to implement the distributed algorithm we used the Akka framework.

2 K-core

2.1 Definition

A k-core of a graph is a maximal connected subgraph in which every node has degree at least k. A node has coreness k when it belongs to a k-core but does not belong to any (k+1)-core. Finally, the coreness of a graph is the maximum coreness of its nodes.

2.2 Algorithm

The sequential algorithm to obtain the k-cores works by recursively removing all nodes which have degree less than k. The algorithm converge when the degree of all remaining vertices is larger or equal to K. Therefore, by iterating on K we obtain the coreness of the nodes and of the graph.

When we start from a graph of which we already calculated the coreness, and we add an edge we can update the coreness values by constructing a candidate set, and checking which nodes have to be updated. The candidate set is built by recursively picking the neighours nodes with equal coreness, and this set is pruned by checking the neighbours with high coreness.

2.3 Distributed algorithm

We can formulate a distributed version of the k-core decomposition [1]. The algorithms consists of two phases.

First, the graph is split into partitions. All partitions are processed in parallel, ignoring the edges between nodes belonging to different partition (frontier edges). This gives a partial coreness value to each node. Afterwards, each frontier edge is sequentially inserted and the coreness value is updated.

In order to speed up the second phase, the frontier edges can be sorted in order to keep each partition disconnected as long as possible, thus allowing the parallel processing of some frontier edges. In order to sort the edges, the master builds a merge tree. Each tree node represents the merging of the two childrens, and it contains the set of all the relevant frontier edges. In this way all the tree leaves can be processed in parallel.

3 Architecture

3.1 Akka actor model

Akka is a general purpose framework [4]to create reactive, distributed, parallel and resilient concurrent applications in Scala or Java on the JVM. Akka uses the Actor model [2] to hide all the thread-related code complexity and gives a simple and helpful interfaces to implement a scalable and fault-tolerant system easily.

Actors are very lightweight concurrent entities. they process messages asynchronously. Each actor is running in isolation from other actors. Furthermore, actors have no shared state, and do not share any resource but rather collaborate by exchanging messages between each other.

Messages between actors are kept to be processed in a FIFO order and delivered at most once. Moreover actors are organized in a supervision hierarchy, so that each actor has a parent that react to his failure by receiving a message.

3.2 Akka cluster

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no failure. It provide this by using gossip protocols and an automatic failure detector. The cluster is able to manage a very large number of actor systems within a single robust system [3].

Akka cluster eagerly sends heartbeats messages to actors in order to keep track of their status. Moreover, the cluster run a leader election algorithm to decide a leader (Master) which will be the coordinator of the other nodes.

The cluster performs additional tasks like node membership events notifications, such as node joining or leaving. When node starts, they need connect only to a seed node, the other members are added to the cluster by gossiping the network status.

4 Project structure

4.1 Cluster nodes

The program cluster can be configured to perform two roles. A single frontend node, holds the master actor which initialize the environment and coordinate jobs between workers. Multiple backends nodes, hosting the worker nodes as well as computing the bulk of the work.

Each backend node is assigned to a WorkerCreator actor that act as a broker on partition loading, and initialize Worker instances. Partition loading request are load balanced between all available WorkerCreators.

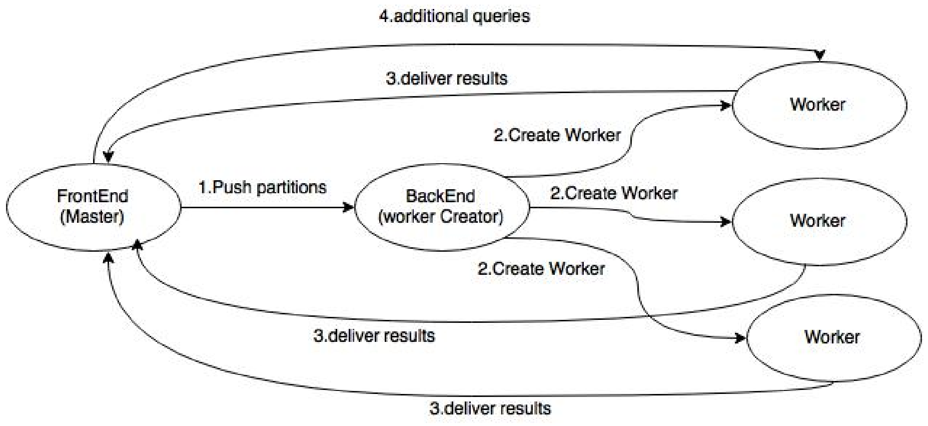


Figure 1: The sequence of messages exchanged between the Master node and the Worker nodes

4.2 Data structures

The algorithm make use of a Graph class that has a set of nodes, represented as integers, and a map from nodes to sets of neighbours in order to represent the edges. As the original input graph is split into multiple parts (subgraph partitions), a worker need a way to identify edges that fall outside of its own partition. Therefore, GraphWithRemoteNodes represents a graph with special nodes that belong to a different partitions.

A GraphWithCoreness is a graph object with an associated coreness value for each node. This class uses the sequential k-shell algorithm in order to compute the partial coreness.

A GraphWithCandidateSet is a graph that is derived from a partition by the relative worker, this object collect a set of candidate nodes. Accordingly, candidate nodes describes a subgraph of all local nodes of the same coreness reachable from the frontier edge node. Further, it includes all neighbours with higher coreness.

The GraphWithCandidateSet graph can have remote nodes (node that belongs to other partitions), so it might be an incomplete graph. In order to compute the full set of nodes to be updated (candidate nodes), the algorithm recursively extract and merge all parts of the external partitions that turns to be relevant (equal coreness).

A PartitionGraph is a graph where each node represent a partition in the original graph, and each edge represent the set of frontier edges between two partitions. This graph is used to build the FrontierEdgeTree, that describes a plan to merge all the partitions.

Each node of this tree waits for the merging operation of its children. The merge is performed on the tree by processing a single frontier edge from each tree leaf. After processing all leaves, the bottom level of the tree is marked as complete and the upper level merges can be performed.

A FrontierEdgeDatabase holds the list of FrontierEdge entries for the master, and manages their status thorough the algorithm.

4.3 Actors

A single Master in the frontend starts by generating the partition files for the workers and contacts the worker creators.

WorkerCreator actors resides inside backends actor systems, one for each node. Their role is only to create on demand Worker instances and pass the loading request to them.

Each Worker loads its own partition data and waits to process Master requests. Once the algorithm is completed all the actor systems are terminated and the cluster is shut down.

4.4 Messages

At the beginning of the process, the master sends a batch of LoadPartition messages (one per partition) to the WorkerCreators in a load balanced way using an adaptive group router provided by Akka standard libraries. These messages contain the name and the means to access the partition data, and allow the worker to obtain their partition as a Graph object.

On the other hand, the workers start processing the partition data, and after it finishes calculating the partial coreness values, it sends a CorenessState message response as a signal.

When all workers have sent this signal, the Master actor starts to update the local frontier edge database by sending CorenessQuery messages. Once it receives enough CorenessReply message such that he knows the coreness of both nodes of a frontier edge, the Master can pull from the Worker the necessary subgraph, using ReachableNodesQuery and ReachableNodesReply.

After collecting the complete reachable nodes graph the Master actor can then perform the pruning operation on the candidate set. Finally it is able to send the NewFrontierEdge messages so that each Worker can add the equivalent remote node to the partition graph and all nodes that need to be updated can have their coreness incremented.

4.5 Behaviour

In a nutshell, the work can be split into four phases. First, each worker calculates the partial coreness of the nodes of the assigned partition, completely ignoring the frontier edge.

Second, the master initializes frontier edge database by querying all frontier nodes (nodes that have a frontier edge) for their coreness value. This value is used to decide which node to use as a starting point for the candidate set.

Third, for each frontier edge the master builds the corresponding candidate set by picking the frontier node with lowest coreness and asking for the reachable nodes subgraph with related candidate nodes.

Fourth, when all candidate nodes have been pulled from the workers, the master can perform the pruning algorithm. It then sends the result with the frontier edge to the workers, that update their local state.

5 Application usage

5.1 Requirements

The project utilizes maven for dependency management. Running the mvn package command, build a single jar file containing all dependencies. To run the jar, only the Java Runtime Environment version 7 or higher is required.

5.2 Launching

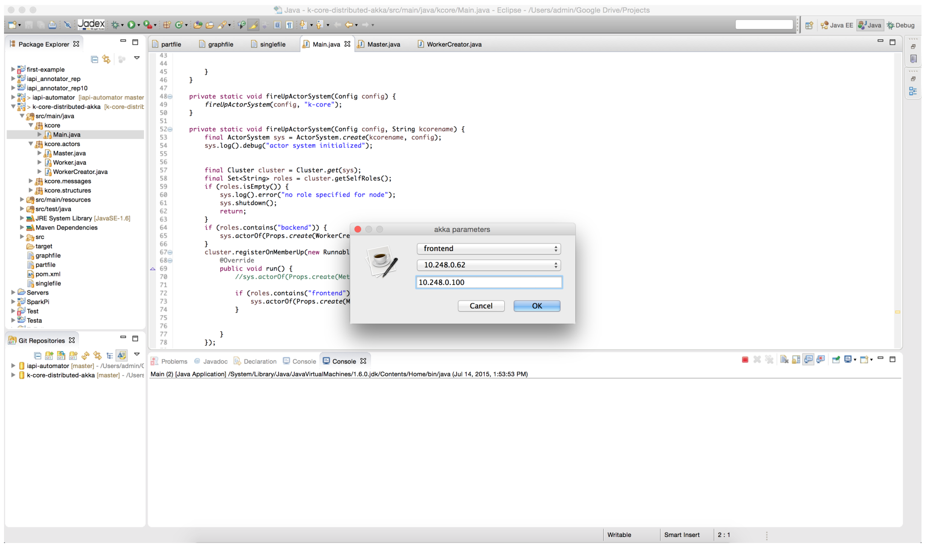
The application can be run in two different ways. It can be run from the command line by giving the configuration file name as the first parameter.

When no command line argument is given, the application starts a simple GUI dialog to set up the network parameters. From this dialog, the user can set the node role, the local IP, and the IP of the seed node. When no configuration is given, the application launches two test nodes using the loopback network.

5.3 Configuration

If the user wants to launch the application from the command line a simple configuration file must be provided. The following is an example of a configuration file:

```
akka.cluster.roles=["$MY_ROLE$"]
akka.remote.netty.tcp.hostname="$MY_IP$"
akka.cluster.seed-nodes=["akka.tcp://k-core@$SEED_IP$:25515",
"akka.tcp://k-core@$MY_IP$:25515"]
```



Java - k-core-distributed-akka/src/main/java/kcore/Main.java - Eclipse - /Users/admin/Google Drive/Projects

```java


        }
    }

    private static void fireUpActorSystem(Config config) {
        fireUpActorSystem(config, "k-core");
    }

    private static void fireUpActorSystem(Config config, String kcorename) {
        final ActorSystem sys = ActorSystem.create(kcorename, config);
        sys.log().debug("actor system initialized");


        final Cluster cluster = Cluster.get(sys);
        final Set<String> roles = cluster.getSelfRoles();
        if (roles.isEmpty()) {
            sys.log().error("no role specified for node");
            sys.shutdown();
            return;
        }
        if (roles.contains("backend")) {
            sys.actorOf(Props.create(WorkerCre
        }
        cluster.registerOnMemberUp(new Runnabl
            @Override
            public void run() {
                //sys.actorOf(Props.create(Met

                if (roles.contains("frontend")
                    sys.actorOf(Props.create(M
                }


            }
        });
```

akka parameters

frontend

10.248.0.62

10.248.0.100

```
Main (2) [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jul 14, 2015, 1:53:53 PM)
```

Figure 2: The simple GUI dialog.

References

[1] Batagelj, Vladimir and Zaveršnik, Matjaž, Fast Algorithms for Determining (Generalized) Core Groups in Social Networks, Adv. Data Anal. Classif., July 2011.

[2] Agha, Gul, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986, Cambridge, Ma, USA.

[3] Cluster Specification, Akka framework documentation, `http://doc.akka.io/docs/akka/snapshot/common/cluster.html`

[4] Akka concurrent, fault-tolerant and scalable framework , Akka documentation, `http://doc.akka.io/docs/akka/snapshot/intro/what-is-akka.html`