
Production Systems and Information Engineering
Volume 7 (2015), pp. 43–53

UTILIZING APACHE HADOOP IN CLIQUE DETECTION METHODS

László Kovács
József Hatvany Doctoral School for Computer Science and Engineering
University of Miskolc, Hungary
Department of Information Engineering
kovacs@iit.uni-miskolc.hu

Gábor Szabó
József Hatvany Doctoral School for Computer Science and Engineering
University of Miskolc, Hungary
Department of Information Engineering
szgabsz91@gmail.com

[Received August 2015 and accepted November 2015]

Abstract. There are many areas in information technology and mathematics where we have to process large graphs, for example data mining based on social networks, route problems, etc. Many of these areas require us to explore the connections among nodes and find all the maximal cliques in the graphs, i.e. all the node sets whose members are mutually connected with each other. One possible and widely used clique detection method is the so-called Bron-Kerbosch algorithm. However, this technique alone might be too slow for big graphs, thus porting the method into a massively parallel system can reduce the overall runtime. This paper introduces some possibilities and starting points in utilizing the open source Apache Hadoop framework that can help in using the resources of multiple computers. The so-called MapReduce architecture makes it possible to divide and conquer the big task into smaller chunks and eventually solve the problem faster than the equivalent sequential methods.

Keywords: graph algorithms, clique detection, mapreduce architecture, Apache Hadoop, parallel systems

1. Applications of Clique Detection

In the era of Internet of Things or IoT [1] for short, sensors are creating unprecedented amount of new data continuously. The measured values can

often be structured into hierarchies or networks. These graphs need to be processed quickly to extract the information out of them, as most of the values are useless in themselves, we need to clean and interpret them, which would be very difficult, often impossible without proper automatization.

However, IoT is only one area that we can apply graph algorithms. A more popular territory of Information Technology where we can work with huge graphs is the databases and services of different social media sites like Facebook, Google+ or Twitter just to mention a few. These sites are excellent examples of world-wide distributed databases that provide almost uninterrupted services 24/7 all around the globe. Facebook Graph API for instance helps us query different parts of their graph structure that contains not only people and their properties, but also places, locations and all the connections among these graph nodes. One of the more popular widely-known area of social network analysis [2] is answering the question *who knows who*, where we can also apply clique detection if we want to find those people who mutually know each other.

A classical mathematical problem, the symmetric travelling salesman problem can also be solved with the help of clique trees [3] which are optimal data structures for storing cliques of a graph. Although this solution requires more advanced mathematical theory as well, clique detection and clique tree building introduces a new method for this well-known problem.

If we want to search for application areas other than IT, we can find many-many methods and models in different disciplines like biochemistry [4]. So we can see in the literature that clique detection is part of many scientific studies, and thus solving it well and quickly is usually crucial. There are, of course, many models that try to achieve the same goal optimally.

One model that we can use deals with neighborhood relationships. The algorithm presented in [5] tries to solve the clique detection problem in the least iteration steps, analyzing the adjacency of the input graph. A more modern approach is given in [6] where genetic programming is used to solve the same problem. As we can expect, genetic programming or any other artificial intelligence method can be utilized in clique detection algorithms, but these tools are often blind or mostly-blind due to the fact that not the specific problem is solved, but the problem is transformed to the domain of the AI model.

One of the classical models is the so-called Bron-Kerbosch algorithm that solves the clique detection problem with classical mathematical tools. This

method was first introduced in [7] and since its first appearance in 1972, many other Bron-Kerbosch variations have appeared that tried to amend the original ancestor in different areas or subproblems.

In this paper the original Bron-Kerbosch algorithm will be used to demonstrate how to apply the old concepts in the fairly new area of MapReduce architecture and Apache Hadoop in order to process huge graphs in parallel.

2. The Bron-Kerbosch Algorithm

Let $G = (V, E)$ be an undirected graph of $v \in V$ nodes and $e = (v_i, v_j) \in E$ edges.

Definition 1 (Clique). *$C(G)$ is a clique of n nodes if* $C(G) = \{v_i \in V \mid 1 \leq i \leq n \wedge \forall v_j, v_k \in C(G) : (v_j, v_k) \in E\}$.

Definition 2 (Maximal clique). *$C(G)$ is a maximal clique if $\nexists C_2(G)$ for which $C(G) \subset C_2(G)$ is true.*

The Bron-Kerbosch clique detection algorithm returns all the maximal cliques for the given graph, effectively returning a set of node sets.

The Bron-Kerbosch method is a recursive algorithm that requires three sets as its input:

- P: the set of nodes that have not been processed yet.
- X: the set of nodes that are not part of the currently investigated clique.
- R: the set of nodes that are all members of the currently investigated clique.

The first step of the algorithm is BronKerbosch$(V(G), \{\}, \{\})$, thus providing the whole graph as the set of yet-to-be-processed nodes.

After every recursive call we check if the current clique is a maximal clique, and if so, we return it. Otherwise we process every member of P and call ourselves recursively, slightly modifying the input sets according to the neighbors of the currently processed node.

Let $N(v)$ be the neighbors of an arbitrary node $v \in V(G)$. With these notations, a simple pseudocode of the Bron-Kerbosch method looks like listing 1.

Listing 1. The Bron-Kerbosch algorithm

```
BronKerbosch(P, X, R):
        if length(P) = length(X) = 0:
                yield R

        for v in P:
                P2 = intersect(P, N(v))
                X2 = intersect(X, N(v))
                R2 = union(R, {v})
                BronKerbosch(P2, X2, R2)
                P = P \ {v}
                X = union(X, {v})
```

As noted above, there are multiple variations on the Bron-Kerbosch algorithm that try to mend it in different ways. Since the method is recursive, by eliminating some of the recursive calls we can make the algorithm faster.

Another way of speeding up the algorithm in case of large graphs is applying parallel programming. In the next sections some of the existing approaches are introduced, then some thoughts on utilizing Apache Hadoop in clique detection using the original Bron-Kerbosch algorithm.

3. Parallel Programming Approaches

In the history of computer science multiple different approaches have come to life in the need of executing code in parallel. The three most popular areas of them nowadays are native threading, GPGPU and distributed systems.

3.1. Native Threading

One of the first methods for parallel execution that everybody considers is native threading. All the currently used desktop operating systems and modern universal programming languages provide a way to fire up a separate thread and load some of the work to that one. Classical example is the mighty POSIX threading [8], but recently C++11 got its own threading types as well, moreover modern languages like Java and C# provide higher level APIs to work with threads like Java's Fork/Join framework and parallel stream API.

However, there are some commonly known drawbacks of using native threading: first of all, the number of parallel threads cannot be higher than the number of logical CPU cores in our system. That's why we need to take care of this fact, because starting more threads can actually slow down our application if we're not cautious. Another issue is that the number of CPU cores is rather low even in the more modern computers, so any threads started above this number will be executed in the old sequential order by the operating system.

3.2. GPGPU

To solve the second issue of native threading, we can utilize graphical processors in our computers to solve general problems. This is called GPGPU [9] which can be used in general purpose applications but is currently applied more widely in computer graphics and games.

One problem with GPGPU is that there are two major uncompatible approaches as well: CUDA [10] which is developed by NVidia and provides a higher level interface to work with the GPU, and OpenCL [11] which is slightly lower level, but platform independent, meaning that it works on both NVidia and AMD graphics cards.

However, we must make sure that the integration of GPGPU in our application is well suited for these APIs, because implementing some algorithms directly on the CPU and some of them on the GPU won't achieve the expected results, as copying data back and forth between the two memory spaces causes a lot of overhead.

3.3. Distributed Systems

The most widely used approach for parallel programming in data mining algorithms – such as our clique detection sample – is using distributed systems, where the parallelism doesn't occur inside a computer, but instead across multiple computers.

With this approach, we eliminate the current hardware limits and build a virtual supercomputer that contains the combined power of the member machines. Of course the delay and overhead cannot be eliminated, but this is not the goal with such systems. The purpose of distribution is to build an elastic and scalable parallel network so that we can add or remove computers

anytime we want.

Although distributed systems have their own history and subcategories, we only deal with the MapReduce architecture and Apache Hadoop in this paper.

4. Apache Hadoop

The main goal of MapReduce [12] and Apache Hadoop [13] is to split the input data to multiple nodes and process them locally. Therefore the distributed system contains mutliple mapper and reducer tasks that can be located on different machines and get input from the core framework.

A mapper task's goal is to produce key-value pairs, grouping the input into multiple subcategories. Each key gets processed by the same reducer task that tries to return an aggregated return value for that key. Data flows and data persistence is controlled by Hadoop that uses the distributed Hadoop File System, or shortly HDFS, that is an open source variant of GFS (see e.g. [14]). HDFS is responsible not only for distributing data to the nodes, but also persisting each chunk on multiple nodes to defend against node failures and data loss. Figure 1 illustrates the logical components and layers of Apache Hadoop 1 and HDFS. Apache Hadoop 2 is slightly different after the introduction of the Yarn framework, but the core concepts have remained the same, only the configuration and flexibility have been improved.

As we can see, Hadoop works in a master-slave fashion where a core component on master controls the slave components. Of course the complexity of the distributed nature and the necessary redundancy is not visible in the rather simplified figure.

5. Using MapReduce to Solve the Classical Bron-Kerbosch Algorithm

When porting a classical algorithm like the Bron-Kerbosch method to the MapReduce architecture, we must solve the following issues:

1. How to split the input data among the nodes?
2. How to test our approach without much overhead?

The next two subsections will answer the above two questions.

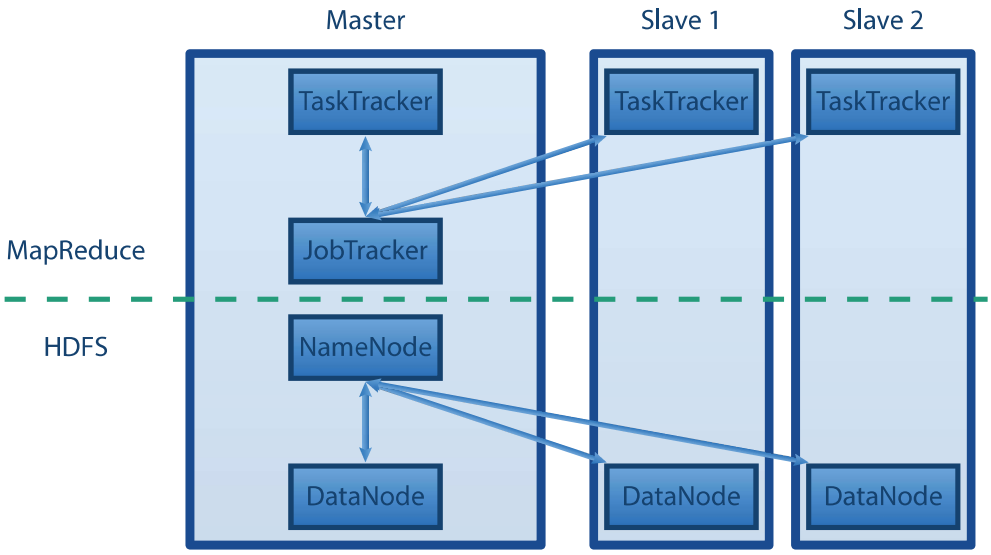

Figure 1. Architecture of Apache Hadoop 1

5.1. Splitting the Input Among the Nodes

When we work directly with Apache Hadoop, we must implement different interfaces that will be called by the framework during execution. These main interfaces are:

- *WritableComparable*: all domain data model classes must implement this interface if they need to be copied among the nodes.
- *Mapper*: this class must be extended by our custom mapper.
- *Reducer*: this class must be extended by our custom reducer.

Every mapper and reducer has four generic type arguments as well: the type of received keys, the type of received values, the type of produced keys and the type of produced values. We have chosen a very intuitive type argument subdivision, that can be seen in table 1.

Table 1. Generic Type Arguments

	Input		**Output**	
	key	**value**	**key**	**value**
Mappers	Null	Graph	Integer	Clique
Reducers	Integer	Clique	Null	Clique

The reason behind this is that the mappers will receive only a graph, it doesn't need to have a key at all. After processing the input graph, they will produce key-value pairs consisting of the hashcode of the resulting clique and the clique itself. The reducer will then receive a clique hash code and the list of every occurrence of the same clique. Since these objects will be equal, the only logic in the reducer is that it returns the first occurrence without a key.

Only one question remains: how to split the big input graph into smaller chunks? Our method was to iterate over the nodes of the graph and extract a smaller graph consisting of the selected node and all of its neighbors. This way we got as many small graphs as there are nodes in the big graph. It's easy to see that if such a small graph contains a maximal clique, it will be a maximal clique in the original graph, too.

Of course there are other approaches for divide and conquer in this example as well. For instance, we could pass the whole graph to every mapper and they would select the appropriate cliques by node index. However, in case of big data this method proved to be slower because the initialization of such distributed systems tend to be slower, since the whole graph has to be copied to every single mapper node, and the mappers must wait until every copy is ready locally.

5.2. Test Environment for Testing Hadoop Based Solutions

When testing a distributed system, usually we have to differentiate a development and a production mode, as maintaining multiple machines for testing purposes is not very remunerative. In our case we had three possibilities beside using a physical distributed system.

The first, and probably the most sophisticated method – that can also be used for production systems – is using a 3rd party cloud service. There are many different providers like Microsoft, Amazon, Oracle, etc. who provide easy-to-use, configurable and scalable solutions. We can pay for any number of virtual machines that we can maintain and use as we wish. However, proof of concepts tend to be more like experimentations that usually take longer time than the available trial of these systems.

Another cheaper method is using local virtual machines. VirtualBox is a free tool that can help us in maintaining arbitrary Linux boxes that we can

use for our test distributed system. Usually one physical machine with enough CPU and memory resources can execute even three or four virtual machines that are more than enough for testing purposes. However, writing automated scripts for VirtualBox is not a very easy task, while maintaining them by hand is a bit error-prone.

A perfect match for our purpose is a relatively new tool called Docker, that also helps us to maintain virtual Linux boxes, however the method of virtualization is faster and more configurable. Moreover, Docker provides a way to write automated scripts that can download, initialize, install and configure systems. Our script uses two custom virtual Debian Linux containers and consists of the following steps:

1. It downloads the base Debian Linux container.
2. It installs all the required software for the machine like Java, Maven, Hadoop, ssh, etc.
3. It copies the required Hadoop configuration files to the appropriate folders based on external XML templates.
4. It builds the test application with Maven.
5. It executes the test application and shuts down the machines.

To keep DRY (don't repeat yourself) principles, two custom Linux Docker containers were used: the first one inherits from the base Debian container and configures a base Hadoop slave machine, while the other one inherits the first one and adds support for Hadoop master nodes. This way the installation steps of Java, Maven, Hadoop, etc. are only stored once.

6. Summary

Hadoop provides an excellent way to make classical algorithms run faster by making them parallel. It provides a way to run parts of the algorithm on different computers, thus eliminating the limits of a single machine. The only thing we must be prepared for, is how we want to divide and conquer the input data. One classical graph algorithm is the Bron-Kerbosch method that returns all the maximal cliques of a graph. The original algorithm is sequential, so we decided to port it to Hadoop by splitting the graph by its nodes. The mappers receive a subgraph as input and return key-value pairs of the hashcode of the found cliques and the maximal cliques themselves. After that the reducers return the first occurrence of the mapper outputs because every reducer receives all the mapper output with the same key (hashcode). To test our proof of concept, we used randomly generated input graphs and Docker as

the virtualization technology because of its scalable and configurable nature. The next step will be to use real physical machines bound in a real cluster and real-life input data, so that we can properly test not just the validity, but the performance of this algorithm compared to the original sequential method.

Acknowledgements

The presented research work was partially supported by the grant TÁMOP-4.2.2.B-15/1/KONV-2015-0003.

REFERENCES

[1] DaCosta, Francis: *Rethinking the Internet of Things: a scalable approach to connecting everything*, 2013, Apress

[2] Scott, John: *Social network analysis*, 2012, Sage

[3] Grötschel, Martin and Pulleyblank, WR: *Clique tree inequalities and the symmetric travelling salesman problem*, Mathematics of operations research, 11, No 4, 1986, pp. 537–569

[4] Butenko, Sergiy and Wilhelm, Wilbert E: *Clique-detection models in computational biochemistry and genomics*, European Journal of Operational Research, 173, No 1, 2006, pp. 1–17

[5] Osteen, Robert E and Tou, Julius T: *A clique-detection algorithm based on neighborhoods in graphs*, International Journal of Computer & Information Sciences, Vol 2, No 4, 1973, pp. 257–268

[6] Haynes, Thomas and Schoenefeld, Dale A: *Clique detection via genetic programming*, 1995, Citeseer

[7] Bron, C and Kerbosch, JAGM and Schell, HJ: *Finding cliques in an undirected graph*, 1972, Technische Hogeschool Eindhoven

[8] Pacheco, Peter: *An introduction to parallel programming*, 2011, Elsevier

[9] Eberly, David H: *GPGPU Programming for Games and Science*, 2014, CRC Press

[10] Cheng, John and Grossman, Max and McKercher, Ty: *Professional Cuda C Programming*, 2014, John Wiley & Sons

[11] Scarpino, Matthew: *OpenCL in Action: How to Accelerate Graphics and Computation. NY*, 2011, USA: Manning Publcations

[12] Dean, Jeffrey and Ghemawat, Sanjay, *MapReduce: simplified data processing on large clusters*, Communications of the ACM, Vol 51,No 1, 2008, pp. 107-113

[13] Wadkar, Sameer and Siddalingaiah, Madhu and Venner, Jason: *Pro Apache Hadoop*, 2014, Apress

[14] Ghemawat, Sanjay and Gobioff, Howard and Leung, Shun-Tak: *The Google file system*, ACM SIGOPS operating systems review, Vol 37, No 5, 2003, pp. 29-43