MapReduce vs. Pipelining Counting Triangles

Edelmira Pasarella[1⋆], Maria-Esther Vidal[3,4], and Cristina Zoltan[1,2]

[1] Computer Science Department
Universitat Politècnica de Catalunya, Barcelona, Spain
[2] Universidad Internacional de Ciencia y Tecnología, Panamá
[3] University of Bonn & Fraunhofer IAIS, Bonn, Germany
[4] Universidad Simón Bolívar Caracas, Venezuela

Abstract. In this paper we follow an alternative approach named *pipeline*, to implement a parallel implementation of the well-known problem of counting triangles in a graph. This problem is especially interesting either when the input graph does not fit in memory or is dynamically generated. To be concrete, we implement a dynamic *pipeline* of processes and an *ad-hoc* MapReduce version using the language Go. We explote the ability of Go language to deal with channels and spawned processes. An empirical evaluation is conducted on graphs of different size and density. Observed results suggest that *pipeline* allows for the implementation of an efficient solution of the problem of counting triangles in a graph, particularly, in dense and large graphs, drastically reducing the execution time with respect to the MapReduce implementation.

1 Introduction

We tackle limitations of the MapReduce programing schema, and devise an alternative computing approach of the *Divide & Conquer* paradigm for solving problems with massive input data. This implementation is based on a *dynamic pipeline* of processes via an asynchronous model of computation, synchronized by channels. To be concrete, we consider the problem of counting triangles. Counting triangles is a *building block* for determining the connectivity of a community around a node, and represents a relevant problem in the context of social network analysis. Indeed, this is, to compute the clustering coefficient which is a measure of interest in social networks. We present both, an implementation of counting triangles based on two rounds of the MapReduce schema proposed by Suri et. al. [5], and the *pipeline* implementation following the approach proposed by Aráoz and Zoltan [1]. In particular, we use Go[5] as the programming language.

Go [2] is a programming language that facilitates efficient implementations of parallel programs, naturally supports concurrency, and implements processes for automatic memory management and garbage collection. Go provides a mechanism of channels to enable the implementation of both *pipeline* and MapReduce, and makes available *goroutines*, which are needed not only for dynamically

⋆ This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R)

[5] https://golang.org/

spawning processes, but also for describing processes that resume their work whenever they stop being blocked. These features are fundamental and crucial for the selection of Go as the programing language for the problem of counting triangles on *pipeline* and MapReduce. We empirically evaluate the performance of *pipeline* and MapReduce on a variety of graphs of different size and density. The observed results although initial, show the benefits of *pipelining* in the implementation of the problem of triangle counting on dense graphs, where the savings in execution time can be of up to two orders of magnitude.

The rest of the paper is organized as follows. In the next section we describe the implementations of the problem of triangle counting in both MapReduce and *pipelining* using the Go language. In Section 3, results of the experimental evaluation are reported and discussed. Finally, in the last section we present the concluding remarks and future work.

2 Implementations of the Problem of Counting Triangles

We present the Go implementation for the algorithms proposed by Suri et al. [5] and by Aráoz et al. [1], for counting triangles in a graph represented as a sequence of unordered edges.

MapReduce Implementation: Go implementation proposed by Suri et al. [5].The program receives as an input a file which is partitioned into as many files as the number of mappers which is set as the number of available cores. In order to reduce the execution time in the MapReduce implementation, the hashing is applied during the Map phase and the mappers communicate via buffered channels with the reducers. The output of the first phase is the set of 2-length paths. Results from the first round are sent to files. The first round output, using the end points as keys, is merged with the set of edges and distributed to the reducers. The output of each reducer is the number of triangles found in its input i.e., the triangles formed by 2-length paths having the same end points and there is an edge between both. A process collects the outputs from the reducers in order to give the final result.

Pipeline Implementation: Go implementation proposed by Aráoz and Zoltan [1]. Corresponds to the composition of a sequence of filters specialized to the vertices of the input graph, and each one works on a set of values that are not consumed by the previous filter. The first filter receives the complete set. Each filter specialize itself with the first incoming edge, using the first node of the edge as responsible node and add the other to an adjacent list. Afterwards, each filter treats the incoming edges, keeping those edges incident to its responsible node and sending the others to its neighbor. The number of filters is equal to the number of classes generated by the relation on the original set[6]. In the implementation, filters are processes/*goroutines* that communicate via unbuffered

[6] The partition relation is created during the execution.

channels and each process is specialized by a responsible node. Further, *goroutines* have three input channels and three output channels. Processes use lists to keep nodes adjacent to the responsible one. Each *goroutine* is specialized to a given node, and utilizes the input of the second channel to collect the edges adjacent to that node. If the edge is not adjacent to the responsible node, the *goroutine* passes the edge to its neighbor. The third channels are used to flow edges into the pipe and each edges is checked if is incident to two nodes adjacent to the corresponding responsible one. In the first channel flow the amount of triangles found by each *goroutines*. A final process adds up the partial results.

3 Experiments

The goal of the experiment is to analyze the impact of graph properties on time and space complexity of both implementations. We study the following research questions: **RQ1)** Is the *pipeline* based implementation able to overcome the *performance* of MapReduce implementation independently of the input graph characteristics?; **RQ2)** Are *density* and *size* of the input graph equally affecting *pipeline* and MapReduce implementations?; **RQ3)** Is the *number of cores* equally affecting *pipeline* and MapReduce implementations? We compare these two implementations using graphs of different densities and sizes. In particular, these graphs are part of the 9th DIMACS Implementation Challenge - Shortest Paths[7]; DSJC.1, DSJC.5, DSJC.9 are graphs with the same number of nodes and different densities, while in Fixed-number-arcs-0.1(FNA.1), Fixed-number-arcs-0.5(FNA.5), and Fixed-number-arcs-0.9(FNA.9) the number of nodes is changed to affect the graph density. USA-road-d.NY and Facebook-SNAP(107)[8] are real-world graphs that correspond to the New York City road network and a subgraph from Facebook, respectively. We consider the execution time (ET) and Virtual-memory (VM) measured in GB. Programs are run on a node of the cluster of the RDLab-UPC[9] having two processors Intel(R) Xeon(R) CPU X5675 of 3066 MHz with six cores each one. The configuration used by us for submitting jobs, is up to 12 cores and 40GB of RAM. Programs are implemented in Go 1.6[10]. The same job is executed 10 times and average in reported, given enough shared memory and a timeout of five hours. Jobs time out at five hours. Graphs with different sizes and densities (0.10, 0.50, and 0.90) are evaluated to study our research questions **RQ1** and **RQ2**. Graphs with high density can be considered as the worst case for both program schemes. Jobs for the *pipeline* program in the different graphs are finished in less than 3 hours, while three jobs of the MapReduce implementations do not produce any response in five hours. The results suggest that the *pipeline* implementation exhibits the best results in response time and virtual memory consumption for graphs as the ones in DSJC.1, DSJC.5, DSJC.9, FNA.1, FNA.5, and FNA.9. Particularly, in the highly dense graphs, i.e., DSJC.9

[7] `http://www.dis.uniroma1.it/challenge9/download.shtml`
[8] `http://snap.stanford.edu/data/egonets-Facebook.html`
[9] https://rdlab.cs.upc.edu/
[10] `https://blog.golang.org/go1.6`

and FNA.9, *pipeline drastically reduces* execution time with respect to MapReduce. Similar performance is observed in the *real-world subgraph* of Facebook (Facebook-SNAP(107)), where *pipeline* execution time overcomes MapReduce by three orders of magnitude. Finally, the graph NY that represents the road network of NY city, is *highly sparse* and the *pipeline* implementation generates a large number of processes that the Go scheduler is not able to deal with.

For the graphs DSJC.1, DSJC.5, DSJC.9, and 107, jobs of the *pipeline* implementation requires less than 18 secs. to be completed and produce the response. Similarly, in graphs DSJC.9, (Facebook-SNAP(107)) and NY, jobs of the MapReduce implementations produce the responses in less than 29 secs. As the obtained results show, jobs for the MapReduce implementation time out at five hours for large graphs: FNA.1, FNA.5, and FNA.9. This negative performance of MapReduce is caused by the *replication factor* of the problem of counting triangles, i.e., the size of the set of 2-length paths (output in the first phase of MapReduce) is extremely large, $O(n^2)$ where n is the number of graph vertices and these graphs have up to 10,000 vertices. These results corroborate our statement that the *pipeline* programming schema is a *promising* model for implementing complex problem and provides an adaptive solution to the characteristics of the input dataset. Furthermore, *pipeline* is competitive with MapReduce and does not require any previous knowledge of the input dataset.

4 Conclusions and Future Work

The well-known problem of triangle counting is utilized to illustrate the features of a *pipeline* implementation as well as the differences with the MapReduce programming schema. Both programs were implemented in multi-processor nodes. The observed results show a superiority in execution time for the *pipeline* implementation even in dense graphs. The only case where MapReduce exhibits a better performance corresponds to a graph where a large number of nodes have an approximate degree of 2, and this particular configuration results in a program that negatively affects the Go scheduler. The results also suggest that the number of processors has a greater positive impact on the *pipeline* implementation than in MapReduce. Based on these results, we can conclude that the *pipeline* schema is highly scalable, and is able to exhibit performance gains on large problem instances with thousands of tasks, seeming to be most promising when a large number of processors work on shared memory. We plan to continue the evaluation of the *pipeline* schema behavior in other complex computational problems, and create a programing framework. Further, other algorithms for counting triangles in graph will be implemented and included in our evaluation study, e.g., algorithms by Hu et. al [3, 4]. However, it is important to highlight that because these algorithms require different representations of a graph, e.g., adjacent lists, and are not implemented as MapReduce, they will require a pre-processing phase and will not be able to be used in graphs dynamically generated. In consequence, the experimental evaluation will have to be redefined in order to conduct a fair comparison of the studied approaches.

Acknowledgements. We thank the staff of the *Laboratori de Recerca i Desenvolupament* (RDlab) of the Computer Science Department of the UPC for their support during execution of the experimental evaluation.

References

1. Julián Aráoz and Cristina Zoltan. Parallel triangles counting using pipelining. `http://arxiv.org/pdf/1510.03354.pdf`, 2015.
2. Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.
3. Xiaocheng Hu, Miao Qiao, and Yufei Tao. Join dependency testing, loomis-whitney join, and triangle enumeration. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 291–301, 2015.
4. Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. I/o-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.*, 39(4):27:1–27:30, 2014.
5. Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.