Comparing MapReduce and *Pipeline* implementations for Counting Triangles

Edelmira Pasarella[*]
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
edelmira@cs.upc.edu

Maria-Esther Vidal[†]
University of Bonn & Fraunhofer IAIS
Bonn, Germany
Universidad Simón Bolívar
Caracas, Venezuela
vidal@cs.uni-bonn.de

Cristina Zoltan
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
Universidad Internacional de Ciencia y Tecnología
Panamá
zoltan@cs.upc.edu

A common method to define a parallel solution for a computational problem consists in finding a way to use the *Divide & Conquer* paradigm in order to have processors acting on its own data and scheduled in a parallel fashion. MapReduce is a programming model that follows this paradigm, and allows for the definition of efficient solutions by both *decomposing* a problem into steps on subsets of the input data and *combining* the results of each step to produce final results. Albeit used for the implementation of a wide variety of computational problems, MapReduce performance can be negatively affected whenever the *replication factor* grows or the *size of the input* is larger than the resources available at each processor. In this paper we show an alternative approach to implement the *Divide & Conquer* paradigm, named *dynamic pipeline*. The main features of *pipeline* are illustrated on 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 evaluate the properties of *pipeline*, a dynamic *pipeline* of processes and an *ad-hoc* version of MapReduce are implemented in the language Go, exploiting its ability to deal with channels and spawned processes. An empirical evaluation is conducted on graphs of different topologies, sizes, and densities. 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

The *Divide & Conquer* paradigm [2] is an algorithm design schema that enables to solve large and complex computational problems in three stages: *i*) *Divide*: an instance of the problem is partitioned into subproblems; *ii*) *Conquer*: the subproblems are solved independently; *iii*) *Combine*: the solutions of the subproblems are combined to produce the final results. The *Divide & Conquer* paradigm is well-known for giving good complexity results. MapReduce [15] is an implementation schema/programming

[*]This research is supported in part 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)

[†]This research is partially supported by the European Union Horizon 2020 programme for the project BigDataEurope (GA 644564).

paradigm of the *Divide & Conquer* paradigm, extensively used in the implementation of complex problems. The *divide* stage is done by establishing an *equivalence relation* on the set of values which are the images of an input set transformed by a *mapping processes*, such that in the *Conquer* stage *reducers* can act on disjoint sets, i.e., each *reducer* acts on a different equivalence class. Finally, other processes collect the *partial results* produced by the reducers to generate the solution in the *combine* stage.

Frameworks that implement the MapReduce schema have had great success and are mostly addressed to run on distributed architectures. Parallelism is a mean for speeding up solutions for computational programs with large amounts of data in memory and that have, in general, a regular behavior. The MapReduce scheme utilizes the Valiant's Bulk Synchronous Parallel (BSP) model of computation [21], and it is defined in terms of a pipe of three steps: *Map*, *Shuffle*, and *Reduce*[1]. Map transforms a domain, where the *equivalence relation* can be established. Shuffle divides the collection into *sub-collections* where the reducers can act independently; the communication between processes in the pipe is done via distributed files which act as shared memory for the processors. Some problems require the composition of several MapReduce processes. The number of composed processes is called the *number of passes* the solution requires. MapReduce implementations require that users provide at least the code for the *Map* and for *Reduce* processes, as well as determine the *number of processors* assigned to the solution. Hadoop [22] is a framework that provides programing file system and operating system abstractions for distributing data and processing; it enables the evaluation and testing the goodness of a MapReduce solution for a problem, as well as facilities the recovery from runtime errors.

The success of the MapReduce schema for solving problems having massive input data has been extensively reported in the literature [14], however, it is also known the MapReduce approach is not suitable for solving problems that require the existence of a shared global state at execution time and solutions that require several passes. In this work, we tackle limitations of the MapReduce programing schema, and present 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 triangle counting[2] and present both, an implementation of counting triangles based on two rounds of the MapReduce schema [20], and the *pipeline* implementation following the approach proposed by Aráoz and Zoltan [1]. In particular, we use Go [8] as programming language. We empirically evaluate the performance of *pipeline* and MapReduce on a large variety of graphs of different size and density. The observed results in this preliminary study suggest that the benefits of *pipelining* in the implementation of the problem of triangle counting on dense graphs and with a large number of edges, where the savings in execution time can be of up to two orders of magnitude.

In summary in this paper, we make the following contributions:

- A comparison of the MapReduce and *dynamic pipeline* programing schemas in the resolution of the problem of counting triangles.
- Implementations in the Go language of two algorithms that follow the MapReduce and *dynamic pipeline* programing schemas to solve the problem of counting triangles. These algorithms exploit the main properties of Go, i.e., channels and spawned processes, and correspond to implementations of the *dynamic pipeline* and MapReduce programming schemas under the same conditions.
- An empirical evaluation of the MapReduce and *dynamic pipeline* based algorithms in a variety of graphs of different density, topology, and size, to demonstrate the performance of both algorithms.

[1] Some implementations combine the shuffle step with the Map step

[2] We consider this problem interesting since it is useful to compute the clustering coefficient which is a measure of interest in social networks

This paper is a revised and extended version of a short paper that was presented at AMW2016[17] and it 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, we present the concluding remarks and future work in Section 4.

2 The Problem of Counting Triangles

2.1 Preliminaries

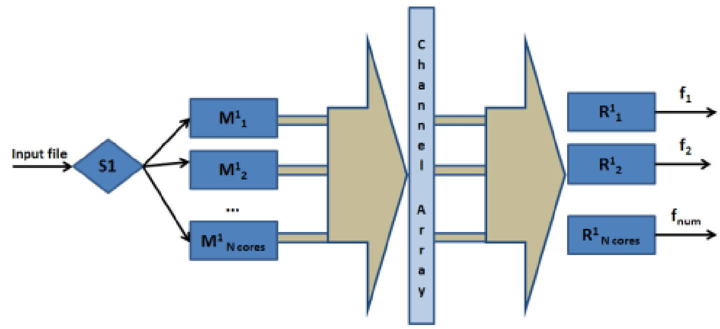
Counting triangles is a *building block* for determining the connectivity of a community around a node, representing a relevant problem in the context of social network analysis. Specifically, given the size of existing social networks, efficiency needs to be ensured, and existing approaches have been exploited the benefits of parallel computation in MapReduce [5, 16, 20], as well as approximate solutions to the problem [4, 12, 18]. We tackle an *exact solution* to the problem, and present two algorithms that exploit the properties of MapReduce and pipeline in the Go programing language. To be concrete, we present Go implementations for the algorithm proposed by Suri et al. [20] and for the algorithm proposed by Aráoz et al. [1], for counting triangles in a graph represented as a sequence of unordered edges. As a precondition, the problem of counting triangles receives undirected simple graphs, i.e., no multiple edges are admitted; to ensure this requirement multiple edges are filtered in a pre-processing stage.

Go [8] is a programming language that facilitates efficient implementations of parallel programs, and naturally supports concurrency as well as processes for automatic memory management and garbage collection. Go additionally provides a mechanism of channels to enable the implementation of both pipeline and MapReduce. Moreover, Go makes available *goroutines*, which are needed not only for dynamically spawning processes, but for describing processes that resume their work when stop being blocked. All 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. In Figures 1 and 2, we illustrate the structure of the two-round MapReduce solution and the *pipeline* approach for the studied problem, respectively. Both implementations are explained in next subsections.

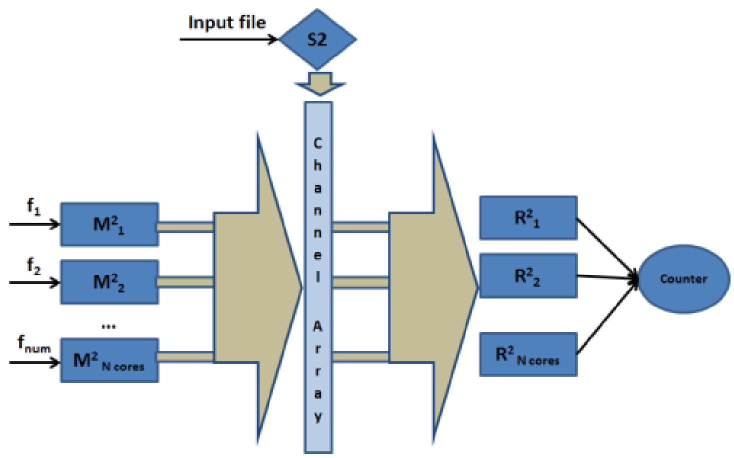
2.2 A MapReduce Solution

Suri and Vassilvitskii [20] present a composition of two MapReduce algorithms for solving the triangle counting problem. In the first MapReduce application, the input is a set of edges of an undirected graph, and the output is a set of 2-length paths having a given responsible node. Each 2-length path with its responsible node is represented by a triple (path-triple). The second application of MapReduce receives the path-triples generated by the previous application of MapReduce and the edge-triples, i.e., the edges present in the input graph with an empty middle element. For each triple, the pair of its end nodes is used as its key. If a path-triple and an edge-triple are in the same cluster, the number of triangles is equal to the cluster size minus one. Otherwise, the number of triangles in the cluster is zero. Adding the number of triangles in each cluster gives the total number of triangles in the graph. It is common that the number of reducers coincides with the number of available processors. So the behavior is not smooth in the number of processors.

MapReduce Implementation: Figure 1 shows the phases of our implementation of Suri and Vassilvitskii's MapReduce algorithm [20]. The program receives as an input a file which is partitioned into as



(a) MapReduce Solution First Round



(b) MapReduce Solution Second Round

Figure 1: **MapReduce Solution for Triangle Counting**. A two-round MapReduce solution: diamonds represent splitting processes and rectangles correspond to mappers and reducers. Thick arrows represent communication through channels

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 and that are sent to files, and then merged with the set of edges, using the process S2, 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.

2.3 A Pipeline Solution

The pipeline solution is the composition of a sequence of filters specialized to the vertices of the input graph, and each one works on a set of values not consumed by the previous filter. Each filter has three inputs and three outputs for receiving/sending messages to their neighbors. The first filter receives the complete set of edges, using the third input. Each filter specializing 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

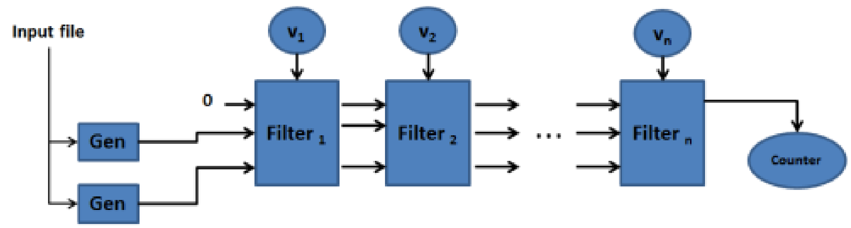


Figure 2: **Pipeline Solution for Triangle Counting**. Rectangles represent filters and ovals correspond to vertices

filter treats the incoming edges, keeping those incident to its responsible node and sending the others to its neighbor. Filters are created dynamically, as new values not consumed by already created filters arrive. As each filter consumes at least one value, no more than $|V|-1$ filters can be created[3]. The number of filters is equal to the number of classes generated by the relations on the original set[4]. Whenever there are no more edges in the third input, the filter will count the number of edges in the second input having both endpoints in the adjacency list of the filter responsible node. After all edges in the second input are processed, each filter has the number of triangles, having the responsible node as one of its nodes. The first input/output of each filters is used to collect the total number of triangles present in the graph. The proof of correctness of this *pipeline* algorithm can be found in [1].

Pipeline Implementation: Figure 2 shows the topology of our implementation of the *pipeline* of the algoritm proposed by Aráoz and Zoltan [1]. In particular, this topology is the composition of a sequence of filters specialized to 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 of edges on the third input. Each new filter specializes itself with the first incoming edge, using the first node of the edge as responsible node and add the other one to an adjacent list. Afterwards, each filter treats the incoming edges, keeping those nodes belonging to edges incident to its responsible node and sending the others to their neighbor.

Whenever all the edges are being processed, each filter has the nodes of the received edges incident to its responsible node. The number of filters is equal to the number of classes generated by the relation on the original set. In the implementation, filters are processes/*goroutines* that communicate via unbuffered 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. In each filter, every incoming edge in the second input is checked if it is incident to two nodes adjacent to the corresponding responsible one. If so, the number of triangles found is increased by one. When there are no more edges, each filter has the total number of triangles in the graph that includes its responsible node. The first channel will carry the number of triangles found by each *goroutines*. A final process adds up the partial results.

3 Experiments

In this section, we present the experimental results of MapReduce and Pipeline implementations for the problem of counting triangles. The goal of the experiment is to analyze the impact of graph proper-

[3] It is an upper bound because the graph does not have isolated nodes
[4] The partition relation is created during the execution.

ties 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*, *topology*, and *size* of the input graph equally affecting Pipeline and MapReduce implementations; **RQ3)** Is the *number of cores* equally affecting Pipeline and MapReduce implementations?. The experimental configuration to evaluate these research questions is as follows:

Graph	# Vertices	# Arcs	Density	File size
DSJC.1	1,000	99,258	**0.10**	1.1MB
DSJC.5	1,000	499,652	**0.50**	5.2MB
DSJC.9	1,000	898,898	**0.90**	9.3MB
Fixed-number-arcs-0.1(FNA.1)	10,000	10,000,000	**0.10**	140MB
Fixed-number-arcs-0.5 (FNA.5)	4,472	10,000,000	**0.50**	138MB
Fixed-number-arcs-0.9 (FNA.9)	3,333	10,000,000	**0.90**	136MB
USA-road-d.NY (NY)	264,346	730,100	**1.04E-5**	13MB
Facebook-SNAP(107)	1,911	53,498	**1.47E-2**	0.524MB

Table 1: **Benchmark of Graphs** Graphs of different sizes and densities. Density is defined as $\frac{\#Arcs}{\#Vertices*(\#Vertices-1)}$

Datasets: We compare these two implementations using graphs of different topologies, densities, and sizes.These graphs are part of the 9th DIMACS Implementation Challenge - Shortest Paths[6]; DSJC.1, DSJC.5, and 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)[13] are real-world graphs that correspond to the New York City road network and a Facebook subgraph, respectively. Table 1 describes these graphs in terms of number of vertices, arcs, graph density, and file size.

Metrics: As evaluation metrics, we consider the execution time (ET) and Virtual-memory (VM). ET represents the elapsed time (in seconds) between the submission of a job and completion of the job including the generation of the final results. VM represents the virtual memory consumed by the batch job measured in GB. Both ET and VM are reported by the *qsub* command when a batch job is submitted to the machine [7].

Implementation: Programs are run on a node of the cluster of the RDLab-UPC[5] having two processors Intel(R) Xeon(R) CPU X5675 of 3066 MHz with six cores each one. The configuration used in the experiments for submitting jobs is up to 12 cores and 40GB of RAM. Programs are implemented in Go 1.6 [9]. The same job is executed 10 times and the average in reported, given enough shared memory and a timeout of five hours.

Discussion 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. Plots in Figures 3 and 4 report on execution time (ET in log10-scale secs) and virtual memory (VM in GB) for each of the schemas. Jobs that time out at five hours are reported using red bars. Jobs for the *pipeline* program in the different graphs are finished in less than 3 hours, while

[5] `https://rdlab.cs.upc.edu/`

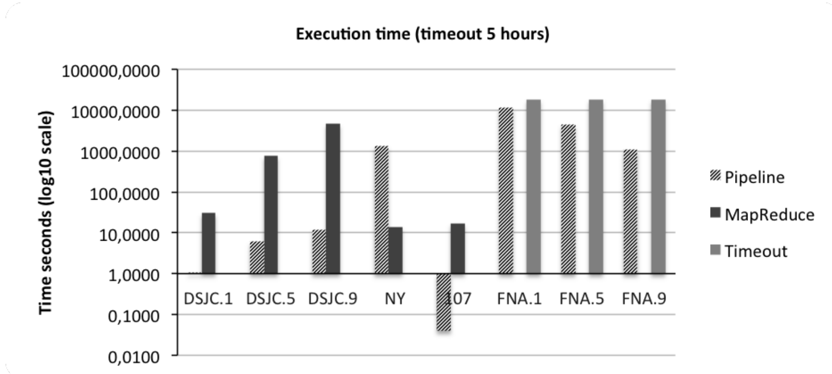


Figure 3: **Execution Time**. For graphs of different size and density, performance of the Pipeline and MapReduce implementations is reported in terms of execution time (ET in log10-scale secs). Jobs that time out at five hours are reported using light grey bars.

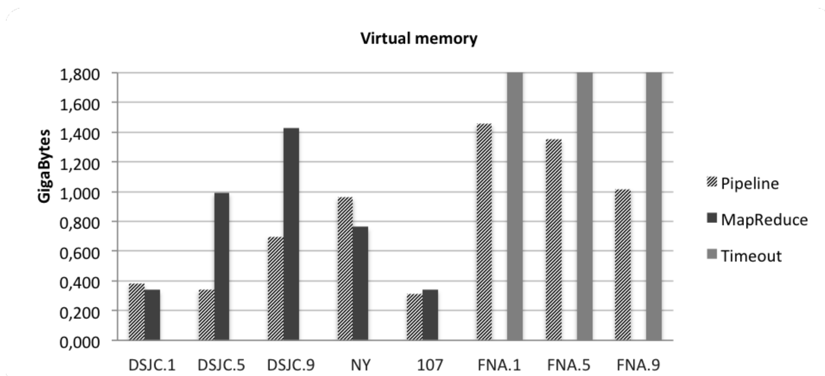


Figure 4: **Virtual Memory**. For graphs of different size and density, performance of the Pipeline and MapReduce implementations is reported in terms of virtual memory (VM in GB). Jobs that time out at five hours are reported using light grey bars.

three jobs of the MapReduce implementations do not produce any response in five hours, i.e., these three jobs time out and are reported in light grey bars in Figures 3 and 4. 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 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.

Our benchmark of graphs is also used to evaluate our research question **RQ3**. Plots in Figures 5 and 6 report on execution time (ET secs) for each of the schemas when the number of cores is eight or twelve. Jobs that time out at five hours are reported using light grey bars. 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

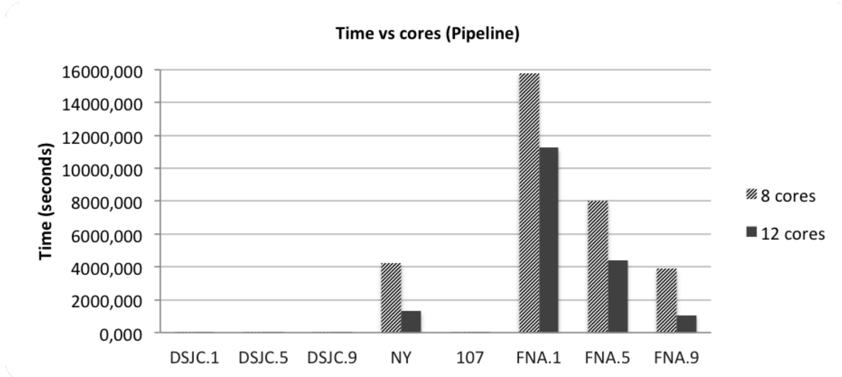


Figure 5: **Impact of the Number of Cores in Pipeline**. For graphs of different size and density, the impact of the number of cores in execution time (ET secs.) of the pipeline implementation is reported. For graphs DSJC.1, DSJC.5, DSJC.9, and 107 execution time is less than 18 secs.

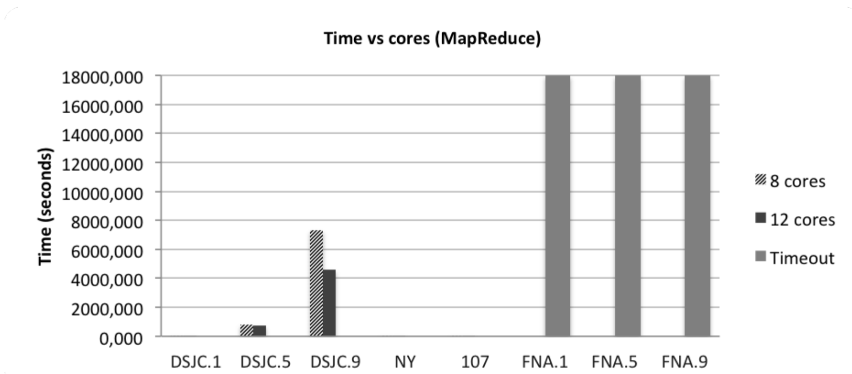


Figure 6: **Impact of the Number of Cores in MapReduce**. For graphs of different size and density, the impact of the number of cores in execution time (ET secs.) of the MapReduce implementation is reported. Jobs that time out at five hours are reported using grey bars. Graphs DSJC.9, (Facebook-SNAP(107)) and NY, jobs of the MapReduce implementations produce the responses in less than 29 secs.

implementations produce the responses in less than 29 secs. As the results reported in Figures 3 and 4, 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

We presented an alternative approach, named *dynamic pipeline*, that follows the *Divide & Conquer* paradigm, and relies on a *dynamic pipeline* of processes via an asynchronous model of computation for process communication. Users of the *pipeline* approach need to provide a sequential code or *filters*, and require no understanding of standard concurrency mechanisms, e.g., threads and fine-grain concurrency control, which are aspects known to be difficult to deal with in order to obtain race condition free code in a parallel solution. Contrary to MapReduce, where implementations differ depending on the architectures, implementations based on the *pipeline* approach can make transparent to users the implementation of channels. The channel abstraction could have several concrete implementations that use shared memory, TCP pipes, or files temporarily persisted in a file system, e.g., as the ones provided by the Dryad distributed technologies[19]. This abstraction can allow for the deployment of the same program in a single machine with several cores, or a net of computing units.

The well-known problem of triangle counting is utilized to illustrate the features of the *pipeline* approach as well as the differences with the MapReduce programming schema. Both programs were implemented in multi-processor nodes. The proposed implementations provide exact solutions for counting triangles by exploiting the main characteristics of the Go programming language, i.e., the evaluation model, a scheduler able to cope with dynamic scheduling, and the notion of channels to enable the communication between processes. The performance of both implementations was empirically evaluated in artificial and real graphs with different sizes and densities. The observed results show a superiority in execution time for the pipeline schema 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 schema than in MapReduce. Based on these results, we can conclude that the *pipeline* approach 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, e.g., in architectures as the one implemented in *The Machine* from Hewlett Packard Labs [6]. In the future, we plan to continue the evaluation of the behavior of the *pipeline* approach in other complex computational problems, and create a programming 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 [10, 11]. 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 & Cristina Zoltan (2015): *Parallel Triangles Counting Using Pipelining*. http://arxiv.org/pdf/1510.03354.pdf. Available at http://arxiv.org/abs/1510.03354.

[6] http://www.labs.hpe.com/research/themachine/

[2] Mikhail J. Atallah & Marina Blanton, editors (2010): *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, 2 edition. Chapman & Hall/CRC.

[3] Jost Berthold, Mischa Dieterle & Rita Loogen (2009): *Implementing parallel Google map-reduce in Eden*. In: *Euro-Par 2009 Parallel Processing*, Springer, pp. 990–1002.

[4] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela & Christian Sohler (2006): *Counting triangles in data streams*. In: *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pp. 253–262.

[5] Jonathan Cohen (2009): *Graph Twiddling in a MapReduce World*. *Computing in Science and Engineering* 11(4), pp. 29–41.

[6] Camil Demetrescu (2010): *9th DIMACS Implementation Challenge*. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[7] Laboratori de Recerca i Desenvolupament (RDlab)-UPC (2016): *RdLab Manual*. ”`https://rdlab.cs.upc.edu/docu/html/manual_cluster/ClusterQuickstart_en.html`”.

[8] Alan A.A. Donovan & Brian W. Kernighan (2015): *The Go Programming Language*, 1st edition. Addison-Wesley Professional.

[9] Go website. Google (2016): *Go version*. `https://blog.golang.org/go1.6`.

[10] Xiaocheng Hu, Miao Qiao & Yufei Tao (2015): *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*, pp. 291–301.

[11] Xiaocheng Hu, Yufei Tao & Chin-Wan Chung (2014): *I/O-Efficient Algorithms on Triangle Listing and Counting*. *ACM Trans. Database Syst.* 39(4), pp. 27:1–27:30.

[12] Konstantin Kutzkov & Rasmus Pagh (2014): *Triangle Counting in Dynamic Graph Streams*. In: *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*, pp. 306–318.

[13] Jure Leskovec (2016): *Social circles: Facebook*. `http://snap.stanford.edu/data/egonets-Facebook.html`.

[14] Jimmy Lin & Chris Dyer (2010): *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers.

[15] Donald Miner & Adam Shook (2013): *MapReduce Design Patterns : building effective algorithms and analytics for Hadoop and other systems*. O'Reilly, Beijing, Kln, u.a. Available at `http://opac.inria.fr/record=b1134500`. DEBSZ.

[16] Rasmus Pagh & Charalampos E. Tsourakakis (2012): *Colorful triangle counting and a MapReduce implementation*. *Inf. Process. Lett.* 112(7), pp. 277–281.

[17] Edelmira Pasarella, Maria-Esther Vidal & Cristina Zoltan (2016): *MapReduce vs. Pipelining Counting Triangles*. In: *To be published in Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management. Panama City, Panama, June 6 - 10, 2016*, CEUR Workshop Proceedings.

[18] Mahmudur Rahman & Mohammad Al Hasan (2013): *Approximate triangle counting algorithms on multi-cores*. In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pp. 127–133.

[19] Dryad Digital Repository (2016): *Dryad*. `http://datadryad.org/`.

[20] Siddharth Suri & Sergei Vassilvitskii (2011): *Counting Triangles and the Curse of the Last Reducer*. In: *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, ACM, New York, NY, USA, pp. 607–614, doi:10.1145/1963405.1963491. Available at `http://doi.acm.org/10.1145/1963405.1963491`.

[21] Leslie G. Valiant (1990): *A Bridging Model for Parallel Computation*. *Commun. ACM* 33(8), pp. 103–111, doi:10.1145/79173.79181. Available at `http://doi.acm.org/10.1145/79173.79181`.

[22] Tom White (2009): *Hadoop: The Definitive Guide*, 1st edition. O'Reilly Media, Inc.