Dynamic Pipeline: An Adaptive Solution for Big Data

Julián Aráoz
Technical University of Catalonia–Barcelona Tech

Edelmira Pasarella
Technical University of Catalonia–Barcelona Tech

Maria-Esther Vidal
Fraunhofer, IAIS, Germany

Cristina Zoltan
Technical University of Catalonia–Barcelona Tech
Universidad Internacional de Ciencia y Tecnología – Panamá

ABSTRACT

The *Dynamic Pipeline* is a concurrent programming pattern amenable to be parallelized. Furthermore, the number of processing units used in the parallelization is adjusted to the size of the problem, and each processing unit uses a reduced memory footprint. Contrary to other approaches, the Dynamic Pipeline can be seen as a generalization of the (parallel) Divide and Conquer schema, where systems can be reconfigured depending on the particular instance of the problem to be solved. We claim that the Dynamic Pipelines is useful to deal with Big Data related problems. In particular, we have designed and implemented algorithms for computing graphs parameters as number of triangles, connected components, and maximal cliques, among others. Currently, we are focused on designing and implementing an efficient algorithm to evaluate conjunctive query.

KEYWORDS

Dynamic Pipeline; Parallelism; Big Data; Concurrency

ACM Reference format:
Julián Aráoz, Edelmira Pasarella, Maria-Esther Vidal, and Cristina Zoltan. 2017. Dynamic Pipeline: An Adaptive Solution for Big Data. In *Proceedings of ACM Celebration of Women in Computing womENcourage 2017, Barcelona, September 2017 (womENcourage 2017),* 1 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Exploiting parallel architecture is a very difficult programing task. There are three basic models of parallelism: task, data, and pipeline parallelism [2]. The Dynamic Pipeline (DP) is a programming pattern that implements a concurrent program that can be easily parallelized. Informally, DP consists of stages connected by (logical) channels. Each stage receives values from upstream via inbound channels and performs some function on that data, usually producing and sending new values downstream via outbound channels. This is, stages can be seen as actors or filters. In a DP pattern, each stage has any number of inbound and outbound channels. The first stage is sometimes called the source or producer; the last stage is the sink or consumer.

Most systems that provide pipeline parallelism employ a construct-and-run model, i.e., programs do not take into account a particular instance of a problem. This is the case for MapReduce [3]. There are other approaches that construct a program using a problem instance, and then run the constructed program to solve the given problem, e.g., Spark[5]. In particular, Spark uses a synchronous computation model.

womENcourage 2017, September 2017, Barcelona
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In a DP, the number of stages depends on the input instance and hence, the pipe construction/destruction takes place at run-time. All the stages are very similar and are normally parametrized by input values. In consequence, to use a DP pattern to solve a given problem stands for defining parameterized actors/filters such that the desired solution flows towards the final output channel. We focus on a particular Dynamic Pipeline pattern: a linear pipeline where each stage receives data from one neighbor and sends data to a single neighbor. This computation model is asynchronous, and synchronized by data availability. We conjecture that all problems solvable using MapReduce can be solved using a linear Dynamic Pipeline pattern. However, contrary to MapReduce which fixes *a priori* a number of processors, a linear DP pattern only exploits the amount of processors required to solve the problem instance.

A simple example to illustrate the use of a DP pattern is the elimination of duplicated values from an input stream, i.e., the output is a duplicated-free stream of values. An actor is a filter in the pipeline and is associated with a value. Actors receive values as input and may produce a value as output. If an actor's associated value and an input value are different, the value is passed to the actor's neighbor; otherwise, the value is eliminated by the actor. An actor is created whenever a value passes through the whole pipeline and it is not filtered by any actor, i.e., an actor is created if and only if an unseen value is received in the input channel.

Given the encouraging results obtained in the implementation of a DP for solving the problem of counting triangles [4], we are working on two main research goals: (a) A formal definition of the DP paradigm. An initial formalization is given in [1]; (b) an experimental framework for benchmarking the performance of very well-known graph algorithms implemented as Dynamic Pipelines. So far, we have implemented pipelines for finding the connected components and maximal cliques of graphs given as streams of arcs. Notice that, in the context of Big Data, the latter characterization of the input graph is an avenue of research worth exploring. Our on-going work is a DP query engine for conjunctive queries.

REFERENCES

[1] Julián Aráoz and Cristina Zoltan. 2015. Parallel Triangles Counting Using Pipelining. http://arxiv.org/pdf/1510.03354.pdf.. (2015). http://arxiv.org/abs/1510.03354
[2] Michael I Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 151–162.
[3] Matthew Felice Pace. 2012. BSP vs MapReduce. *CoRR* abs/1203.2081 (2012). http://arxiv.org/abs/1203.2081
[4] Edelmira Pasarella, Maria-Esther Vidal, and Cristina Zoltan. 2017. Comparing MapReduce and pipeline implementations for counting triangles. *Electronic proceedings in theoretical computer science* 237 (2017), 20–33.
[5] Spark 2017. Apache Spark. (2017). http://spark.apache.org/