ProcessFast, a Java framework for the development of concurrent and distributed applications

Andrea Esuli and Tiziano Fagni

Istituto di Scienza e Tecnologie dell'Informazione
Consiglio Nazionale delle Ricerche
56124 Pisa, Italy
E-mail: {firstname.lastname@isti.cnr.it}

Abstract. Today, any application that requires processing information gathered from the Web will likely require a parallel processing approach to be able to scale. While writing such applications, the developer should be able to exploit several types of parallelism paradigms in a natural way. Most of the available development tools are focused on just one of these parallelism types, e.g. the data parallelism, stream processing, etc. In this paper, we introduce ProcessFast, a Java framework for the development of concurrent/distributed applications, designed to allow the developer to integrate both stream/task parallelism and data parallelism in the same application and to seamlessly combine solutions to sub-problems where each solution exploits a specific programming model.

1 Introduction

Most of the frameworks for parallel and distributed computing focus on a single parallel computing paradigm, e.g., stream processing [6, 1, 5], map-reduce [3, 7, 8], task parallelism [4]. Complex applications could benefit from using a combination of different approaches. For example, a text mining system that classifies a stream of tweets can be decomposed into a set of parallel tasks (crawling, NLP processing, indexing, classification, aggregation, report) connected via streams, with each task possibly exploiting data parallelism to efficiently apply the same computation to batches of data. Implementing such a system usually requires to combine several frameworks, with the added burden of implementing a communication layer among them. Moreover, the target architecture of the system, e.g., a single multi-core machine or a distributed environment, will result in different choices of frameworks, since each framework is usually targeted to produce its maximum efficiency on a specific architecture. Changing the runtime architecture will often require to change the underlying parallel computing framework[1], with non trivial implementation cost.

[1] Many tools have a standalone mode that simulates a distributed environment on a single machine, but it is mainly thought as a development tool, not for production use.

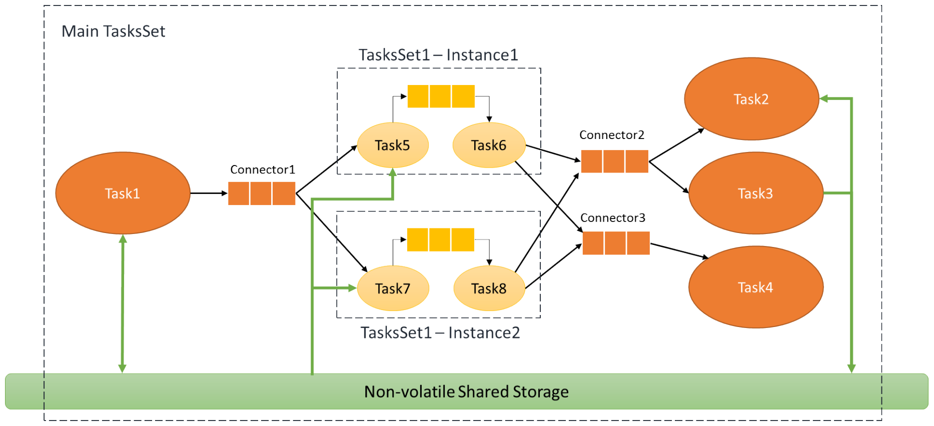

Fig. 1. Schema of a ProcessFast application

We are developing ProcessFast (PF), a Java framework that aims at providing a seamless integration of different parallel computing models, by combining the functionalities provided by different parallel processing frameworks into an homogeneous API. PF does not aim at implementing yet another parallel computing framework, its purpose is to act as a higher-level API that abstracts the functionalities of current parallel computing frameworks, allowing to write scalable applications that once developed can be deployed, and executed efficiently, on different architectures by only switching the PF runtime layer that implements the API on the better suited frameworks. This paper introduces the PF main concepts, structures, and functionalities. The current status of the development consists of the PF API and a first implementation of the API on a single machine architecture mainly based on wrapping the GPars library [2].

2 ProcessFast, an overview of the programming model

The PF API[2] defines how the developer can write applications that use and combine task/stream parallelism and data parallelism. It provides a lock-free programming model in which an application can be defined in terms of a set of asynchronous processes that are able to intercommunicate.

2.1 TaskSet, a logical container and a reusable "black box"

As shown in Figure 1, the topology of an application is defined inside a *TaskSet*. A TaskSet is a logical container of *Tasks* (detailed in Section 2.2) and *Connectors* (detailed in Section 2.3). A TaskSet is also a Task, and thus can be used as a "black box" inside another TaskSet. A TaskSet allows to implement *task/stream parallelism* through the use of Connectors to let the contained Tasks communicate together. *Barriers* can be used to synchronize the execution of Tasks.

[2] ProcessFast public repository: https://github.com/tizfa/processfast-api

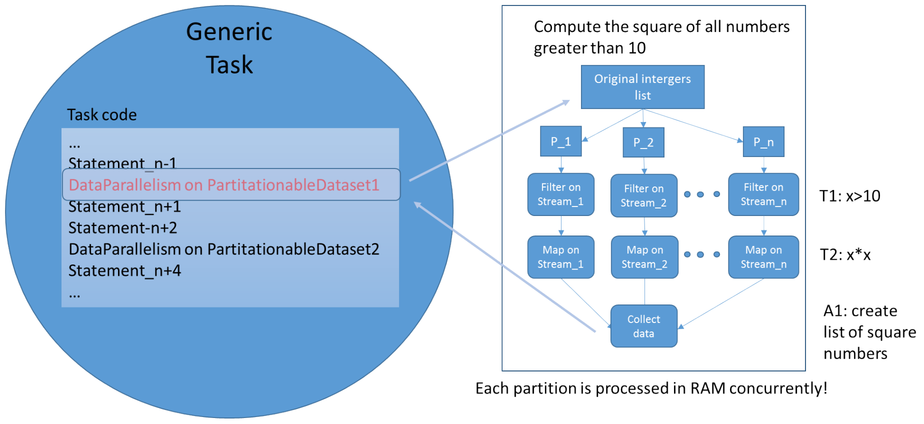

Fig. 2. The internal structure of a Task

2.2 Task, a stateful and asynchronous logical process

A *Task* is the minimal unit of execution in PF. Every Task is a stateful logical process which runs asynchronously with respect to the other tasks in the application and can be created in multiple instances inside a single program. A Task is able to communicate through Connectors only with the other Tasks defined in the same TaskSet, or externally using the input and output Connectors of the parent TaskSet (which define the entry and exit points of the TaskSet taken as a black box). As shown in Figure 2, a Task internally can exploit the *data parallelism* by operating concurrently on *PartitionableDatasets* (PDs). The concept of PD is very similar to that of RDD in Spark [8]. A PD is a read-only data structure which can be split in n partitions, where every partition can then be processed concurrently as a data stream. PDs can be processed by applying two types of operations, following a map-reduce model:

– *transformations*: each item in the input stream is transformed in some way resulting in a new item in the output stream (e.g. T1 and T2 in Figure 2).
– *actions*: items from input data stream are collected and processed to produce an aggregated result that is returned to the caller task (e.g. A1 in Figure 2).

2.3 Connectors, interprocess communication based on queues

A Connector is a shared queue, belonging to a TaskSet, uniquely identified by a name. Connectors can have single or multiple Tasks attached as readers and writers. A Task can consume exclusively an item read from a Connector (*first come, first served*, data parallelism) or the Connector can provide to each reading Task the same complete sequence of items as written to the connector (*broadcasting*, task parallelism). Write operations are generally asynchronous (a Task after posting a message on a specific connector can continue its computation) while the read are synchronous and blocking, though is it also possible to define synchronous read/write connections.

2.4 Shared permanent storage

The PF API defines a shared permanent storage which allows direct access to some high levels data structures. The main purpose of the permanent storage is to reduce the amount of information transmitted through connectors and to rely instead on solutions that are best fit for the target architectures. The supported data structures are:

- *Array*: a direct access unidimensional array. The structure supports PD views, thus it is ready for parallel processing.
- *Matrix*: a direct access bidimensional array. The structure supports PD views, allowing parallel processing by rows or by columns.
- *Dictionary*: a key/value collection.
- *DataStream*: a byte stream used to load/store data.

3 Conclusions

We have introduced the main ideas behind PF, whose grand goals are (i) to allow developers to seamlessly integrate different parallel computing models into their applications, and (ii) to implement a *write once, run (efficiently) anywhere* model for parallel/distributed applications. We have recently completed the first implementation of the ProcessFast API based on Groovy GPars library [2]. One of our first tests, on a eight cores CPU, obtained a five-fold improvement against a sequential implementation of matrix multiplication using matrixes with a size of 10'000 by 10'000. Future development will focus on implementing the API on a distributed architecture and on running comparative tests with the well-known alternatives (e.g., Hadoop, Spark, Storm).

References

1. Apache Samza. http://samza.apache.org/.
2. Gpars: Groovy parallel system. http://gpars.codehaus.org/.
3. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
4. J. Dinan, S. Krishnamoorthy, D.B. Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 586–593, Sept 2008.
5. Gianmarco De Francisci Morales and Albert Bifet. Samoa: Scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16:149–153, 2015.
6. Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.
7. Tom White. *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.
8. Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.