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

Tiziano Fagni, ISTI-CNR, Pisa
Andrea Esuli, ISTI-CNR, Pisa

IIR 2015. Cagliari, May 25-26 2015

* Sorry, not today ☺

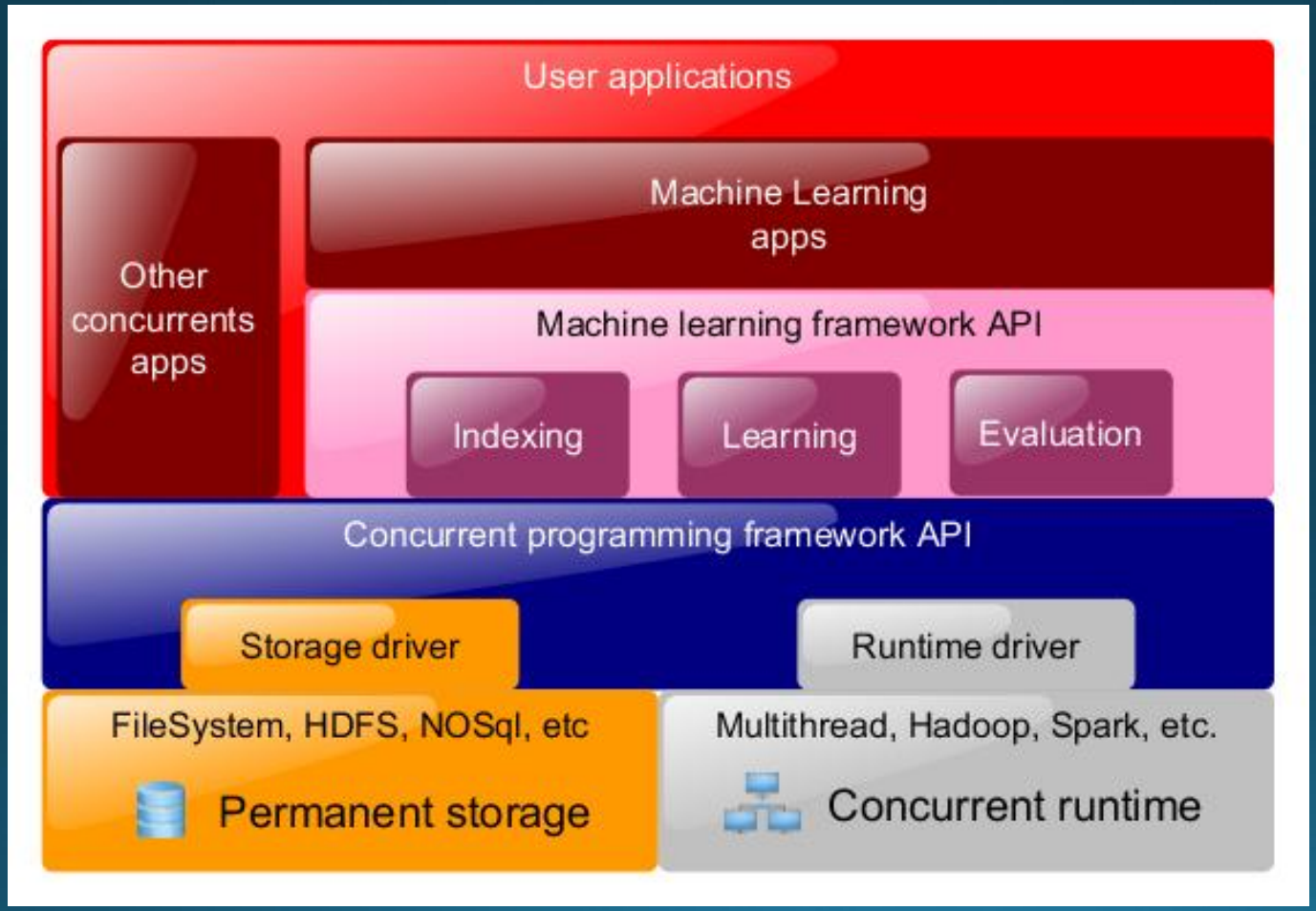
Tools for concurrent programming: state of the art and current limitations

- A lot of tools available specialized for
  - Data-parallelism (Hadoop, Spark, etc.)
  - Stream-parallelism (SAMOA, Storm, S4, ecc)
  - Task-parallelism (MPI, Akka, etc.)
- Tools generally impose a specific programming model and are not flexible.
- Difficult to integrate both task/streaming-parallelism and data-parallelism.
- Most tools are optimized for distributed computing on a lot of data. What if
  - I have available few computational resources?
  - I have to analyze few gigabytes of data?

Overcome current limitations: our proposal for concurrent programming

- A new framework for developing concurrent applications
  - Seamless mix of task/streaming-parallelism and data-parallelism.
  - Quite simple API, but powerful enough to write complex software.
  - Embrace «write once, deploy anywhere» motto .
  - The API virtualizes access to high level data structures on permanent storage (hiding distribution, redundance…).
- A first specific implementation of the runtime optimized for multithread on a single machine.
- A distributed version of the runtime using the existing state-of-the-art software stack (Hadoop, Spark, Storm, etc).

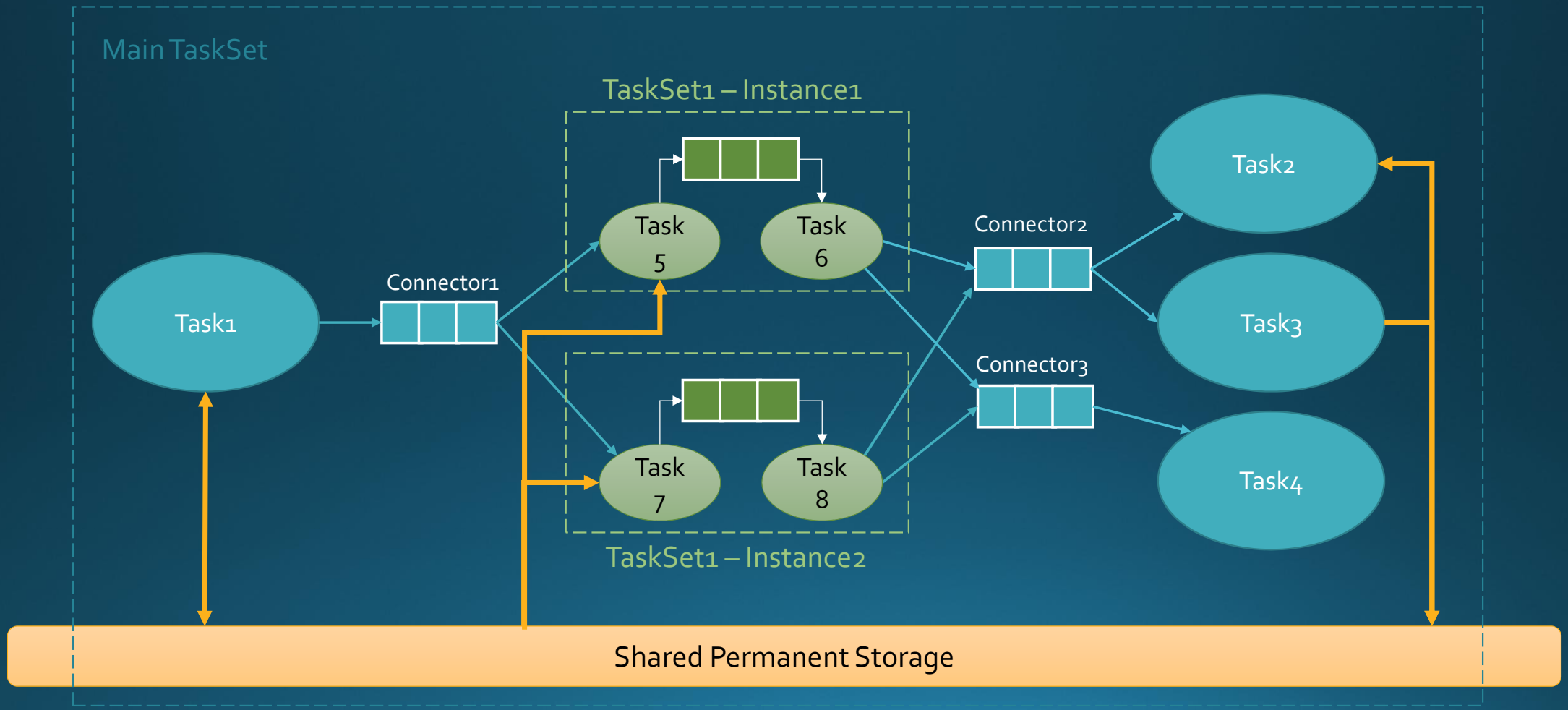
High level view of software architecture



ProcessFast-api, a reference specification for the new framework

- Express computations
  - It allows to express stateful asynchronous tasks communicating using named queues.
  - Tasks synchronization is expressed by named barriers.
  - For each task you can express its parallelism degree.
  - Data parallelism inside each task through the use of «partitionable dataset»
    - Tranformations (map, filter, etc.)
    - Actions (collect, count, etc)
- Transparent access to high level structures on permanent storage
  - Array
  - Matrix
  - Dictionary
  - Data stream

ProcessFast: programming model

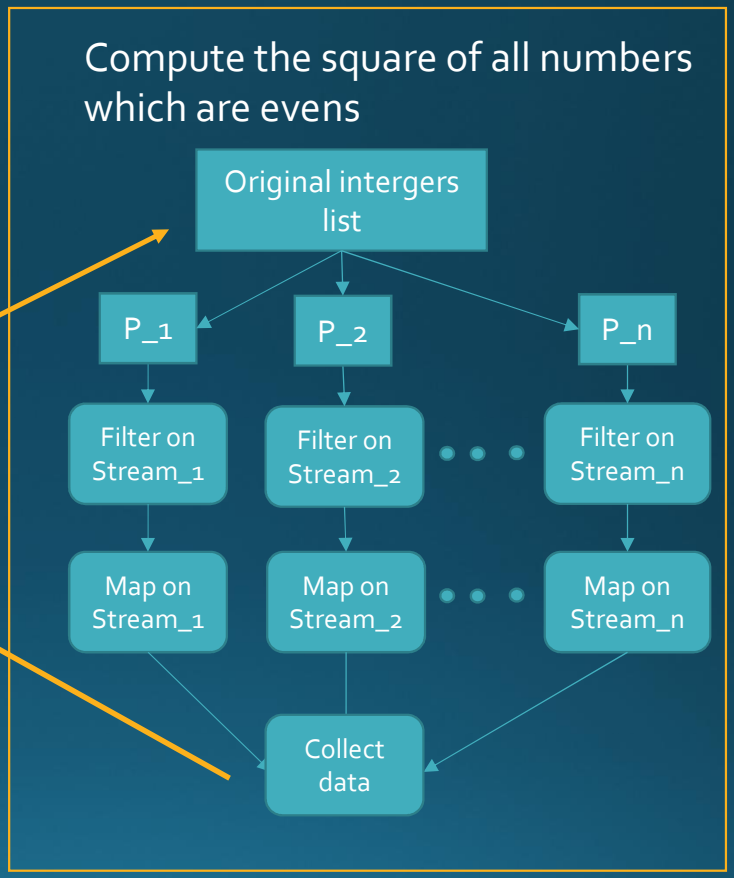


ProcessFast: programming model (2)

Generic Task

Task code

```
...
Statement_n-1
pd.filter((tdc,item)->item % 2 == 0)
  .map((tdc, item)->item*item)
  .collect()
Statement_n+1
Statement-n+2
Statement_n+4
...
```



T1: x % 2 == 0

T2: x*x

A1: create list of square numbers

Each partition is processed in RAM concurrently!

API: details about a Connector

- Each connector is a shared queue with multiple readers/writers.
- Each connector can have different sematic. There are the following:
  - LOAD_BALANCING_QUEUE: write and read multiple times. The data is distributed among the readers.
  - BROADCAST_QUEUE: write and read multiple times. Each reader gets the same set of messages, in the order as they had been wrote on the queue
  - **TO_ADD:** KEY_QUEUE: write and read multiple times. Each reader has access to data with a specific key.
- The write operations are asynchronous, the read are synchronous.
- Possibility of synchronous write_and_get operations

API: details about a Task

- A Task is a stateful logical asynchronous process to be executed by the system.
- The programmer can specify the min and max number of instances of the task
- Each task type has a relative importance to other tasks in tasks set in computational resources allocation for data parallelism
- For each task instance, the programmer can specify
  - The logical vm where should execute the task instance
  - The name of the task
  - A data dictionary to specify specific parameters for the instance
  - Which input/output connectors (virtual or real) the instance must have access on
  - Which barriers (virtual or real) the instance must have access on

API: details about a TaskSet

- As a container
  - Declare tasks or tasks sets
  - Declare connectors and barriers
- As reusable «black box» to provide pluggable high level functionalities
  - Specify the number of instances
  - For each instance, the programmer can specify
    - The logical vm where should execute the taskset instance
    - The name of the taskset
    - A data dictionary to specify specific parameters for the instance
    - Which input/output connectors (virtual or real) the instance must have access on
    - Which barriers (virtual or real) the instance must have access on

ProcessFast odd-even example

```
public class OddEvenExample {
    public static void main(String[] args) {
        GParsRuntime runtime = new GParsRuntime();
        TaskSet ts = runtime.createTaskSet();
        ts.createConnector("DISTRIBUTOR", ConnectorType.LOAD_BALANCING_QUEUE);

        ts.task((TaskContext tc) -> {
            ConnectorWriter dist = tc.getConnectorManager().getConnectorWriter("DISTRIBUTOR");
            for (int i = 0; i < 10000; i++)
                dist.putValue(new Random().nextInt(1000));
            dist.signalEndOfStream();
        }).withConnectors(wci -> {
            wci.getConnectorManager().attachTaskToConnector(wci.getTaskName(), "DISTRIBUTOR", ConnectorCapability.WRITE);
        });

        ts.task((TaskContext tc) -> {
            ConnectorReader dist = tc.getConnectorManager().getConnectorReader("DISTRIBUTOR");
            while(true) {
                ConnectorMessage cm = dist.getValue();
                if (cm == null)
                    break;
                int val = (int) cm.getPayload();
                if (val % 2 == 0)
                    tc.getLogManager().getLogger("TEST").info("The number "+val+" is odd");
                else
                    tc.getLogManager().getLogger("TEST").info("The number "+val+" is even");
            }

        }).withConnectors(wci -> {
            wci.getConnectorManager().attachTaskToConnector(wci.getTaskName(), "DISTRIBUTOR", ConnectorCapability.READ);
        }).withNumInstances(4, 4);
        runtime.run(ts);
    }
}
```

ProcessFast vs other popular systems

- Hadoop
  - Very popular and with a very big software ecosystem
  - Provides map-reduce model, heavy use of I/O, poor performance in iterative computation, quite complex to configure
  - Standalone mode just for debugging, single node installation very slow, no streaming support.

- Spark+SparkStreaming
  - Very popular and with a big software ecosystem (Mlib, SparkSQL, etc.)
  - Batch processing with RDDs, streaming via RDDs mini-batches, good performance, easy and fun to use.
  - Streaming: only continuos streaming, state sharing not easy, no control or explicit synchronization among processes.

ProcessFast vs other popular systems (2)

- Apache Storm
  - Very popular in BIG DATA integrated solutions
  - Realtime SPE based on concepts of «spouts» and «bolts», very simple and flexible API, continuos streaming
  - Trident adds high level abstraction on Storm, supporting exactly-once processing and making stefeful processing easier (similar to SparkStreaming)

A preliminary set of experiments (1)

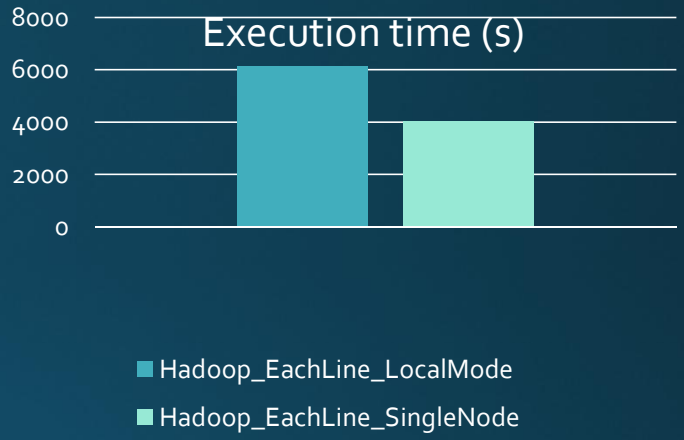
- Intel(R) Core(TM) i7 CPU X 980 @ 3.33GHz, 12 cores + SATA Disk (about 125 MB/s in read operations)
- AMD FX3850 8 cores + SSD disk

Test's goal: analyze 10 Gb of textual data and extract all words occurences in the text.

Software used

- ProcessFast latest snapshot: https://github.com/tizfa/processfast-api
- Apache Hadoop 2.7.0
- Apache Spark 1.3.1

A preliminary set of experiments (2)



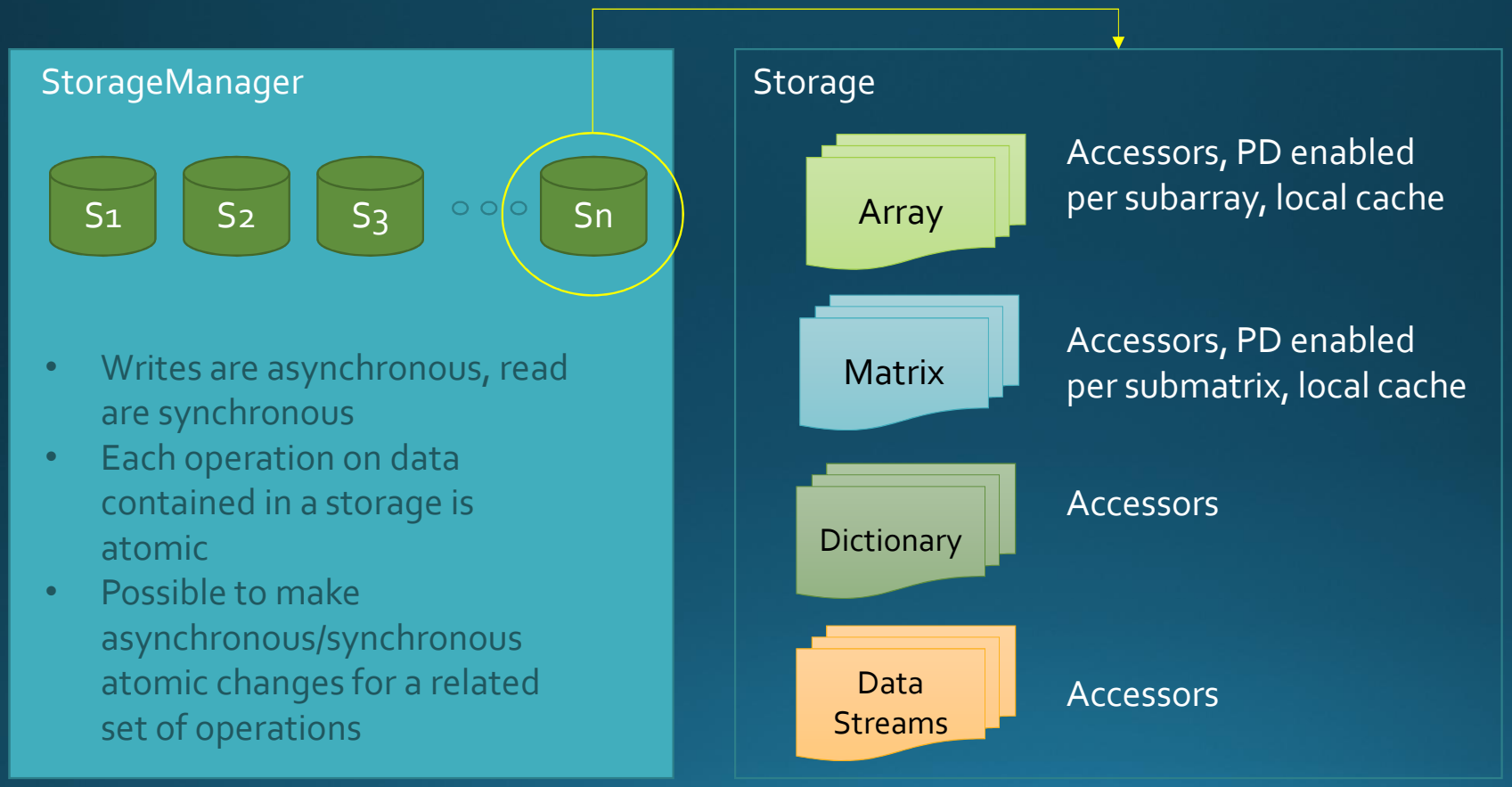
To be done….

- Improve ProcessFast current runtime implementation: Groovy resulted suboptimal.
- Complete the implementation of the defined API
- Enrich our set of experiments with practical scenarios (e.g. machine learning apps)
- Add other systems to this comparison (Storm+Trident, Samoa, etc.)
- Evaluate quality of existent systems on restricted environments
  - Can one of such tools be used effictively? Which are the weakest points? It is better to use one of these or use your own software stack?

Thanks, that'all!

Questions?

The shared permanent storage



- Writes are asynchronous, read are synchronous
- Each operation on data contained in a storage is atomic
- Possible to make asynchronous/synchronous atomic changes for a related set of operations

Data-parallelism by using PartitionableDataset

- Idea taken from Spark with its RDDs
- A PartitionableDataset (PD) is a read-only collection of data which can be divided in disjoint partitions and processed in parallel
- A PD can be modified by tranformations (e.g. map, union, intersection, filter, dictinct, etc.)…
- and the results can be obtained by using actions (count, collect, reduce, etc.)
- Operations on PD can be tuned by
  - Enabling/disabling local computation
  - Sizing partitions
  - Caching and reusing partial results

Data-parallelism by using PartitionableDataset: simple example

```
ts.task { tc ->
    // Other code...

    // Build a random list of numbers.
    Random r = new Random()
    def l = []
    (1..10000000).each {
        l << r.nextInt(1000)
    }

    // Create a partitionable dataset of the list.
    def pd = tc.createPartitionableDataset(new GenericIteratorDataSourceIterator<Integer>(l.iterator()))

    // Apply some transformations and cache it...
    def squareResults = pd.enableLocalComputation(true)
          .filter { tdc, item -> item.v2 % 2 == 0 }
          .map { tdc, item -> item.v2**2 }
          .cache(CacheType.RAM)

    // Apply some actions to obtain something back.
    int numItems = squareResults.count()
    List<Integer> first100 = squareResults.take(0, 100)

    // Other code...
}
```

Storage manager: API usage example

```
// Asynchronous write.
tc.storageManager.getStorage("storage").getArray("array").setValue(10, 4.5)

// Synchronous read.
double v = tc.storageManager.getStorage("storage").getArray("foo").getValue(10)

// Enable local cache of items 0--1000 .
tc.storageManager.getStorage("storage").getArray("array").enableLocalCache(true, 0, 1000)

// Asynchronous atomic operations.
tc.storageManager.atomic { ctx ->
    def m = ctx.storageManager.getStorage("storage").getMatrix("foo")
    m.setValue(0, 1, m.getValue(0, 1) + 1)
    def m2 = ctx.storageManager.getStorage("storage2").getMatrix("foo2")
    m2.setValue(0, 1, m.getValue(0,1)**2)
}

// Asynchronous atomic get operations by using a future.
def results = tc.storageManager.atomicGet { ctx ->
    def m1 = ctx.storageManager.getStorage("storage").getMatrix("foo")
    def m2 = ctx.storageManager.getStorage("storage2").getMatrix("foo2")
    new RamDictionary().put("res", m1.getValue(0,1)*m2.getValue(0,1))
}
```

Hardware failures handling in ProcessFast

- Data-parallelism through PD: the runtime automatic handle individual sub-tasks failures (Spark docet!)
- Tasks(streams)-parallelism: the programmer can use data snaphots to recover the execution of a program from a specific point in time.
  - Methods available: loadLastCheckpoint(), saveCheckpoints() and deleteLastCheckpoint()
  - Can use barriers to synchronize multiple tasks
  - Only available in a executable tasks set

ProcessFast: current status

- Java API 1.0.0 near-stable and complete
  - API documentation is quite good.
  - There is no developer's guide!
- ProcessFast-gpars, an implementation of the runtime based on Groovy and Gpars library
  - Tasks/streaming parallelism almost complete
  - Data parallelism through PDs is quite complete
  - To be developed: data snapshotting, PDs permanent partial results
- Permanent storage: partly developed using FoundationDB NoSQL system

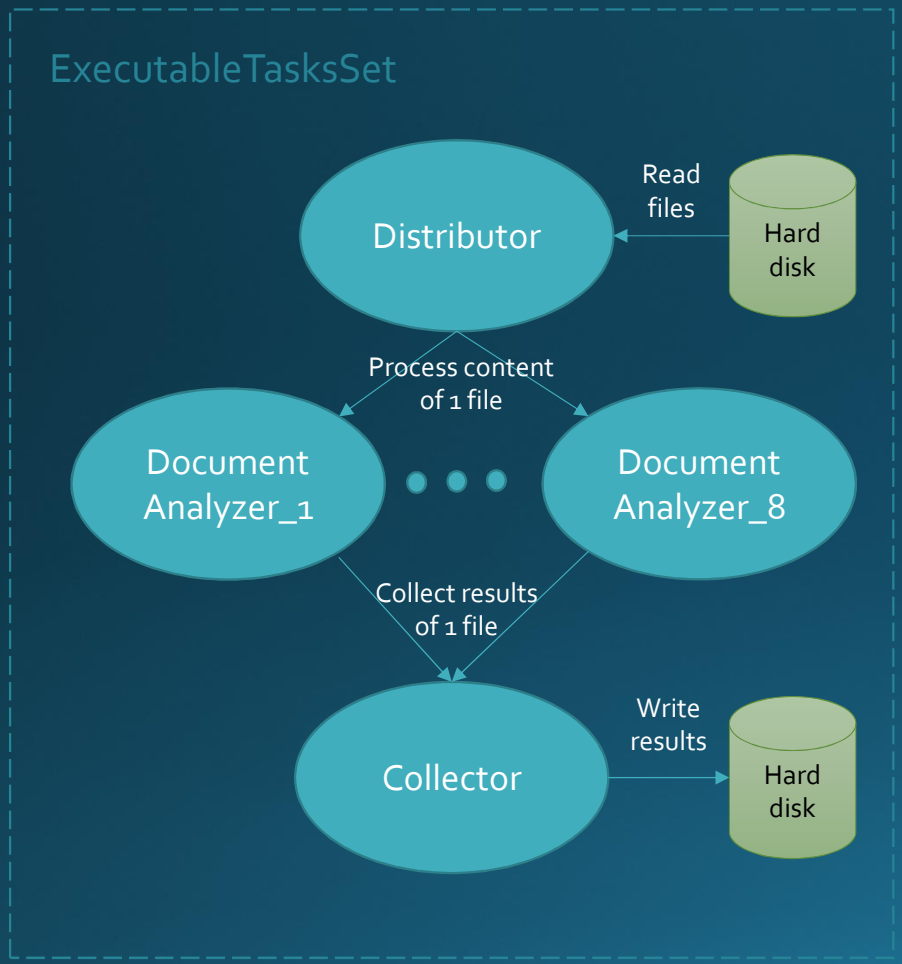
Example1 – Tweets analyzer

- Count in real time the average length of tweets and the average number of hashtags for tweets in english containing the word «apple»

Example2 - Documents analyzer, sequential version

- Collection of 1006 files, each about 10 Mb in size and containing on average about 1300 english Wikipedia articles already in txt format.
- Get the occurrences of each word, excluding words with length <= 3 and number of occurrences < 3
- Test machine
  - Processor: AMD FX-8350 4.1 Ghz, 8 cores
  - 32 GB RAM
- Sequential version
  - Groovy code length: 97 lines
  - Execution time for all files: 14m 40s

Example2 - Documents analyzer, stream version



- Distributor queue has max size of 50 items
- 8 instances of DocumentAnalyzer (same sequential code)
- Groovy code length: 144 lines without hadware failure management
- Execution time: 3 m 15s

Example2 - Documents analyzer, PD version

Main task

```
// Read file list...
def provider = readFilesList()

def pd = tc.createPartitionableDataset(provider)
def mapWords = pd.withPartitionSize(50)
   .map { tdc, Pair<Integer, String> item ->
       Map<String, Integer> mapWords =
analyzeSingleFile(item.v2)
       return mapWords
    }.reduce { tdc, map1, map2 ->
       HashMap<String, Integer> mapWords =
combineTogether(map1, map2)
       mapWords
    }
// Write results...
```

- Use a map-reduce approach
- On GParsRuntime, process 50 files at a time
- Use a threads pool of size 8
- Groovy code length: 109 lines with automatic hardware failure management
- Execution time: 3 m 28s

Example3 – Product of matrices

- Given matrix A (10000 x 100) and matrix B (100 x 10000), the program computes C=A x B.
- Sequential version: about 50 s
- Parallel version with PDs and threads pool of size 8: about 10 s