www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based programming in COMPSs to converge from HPC to Big Data

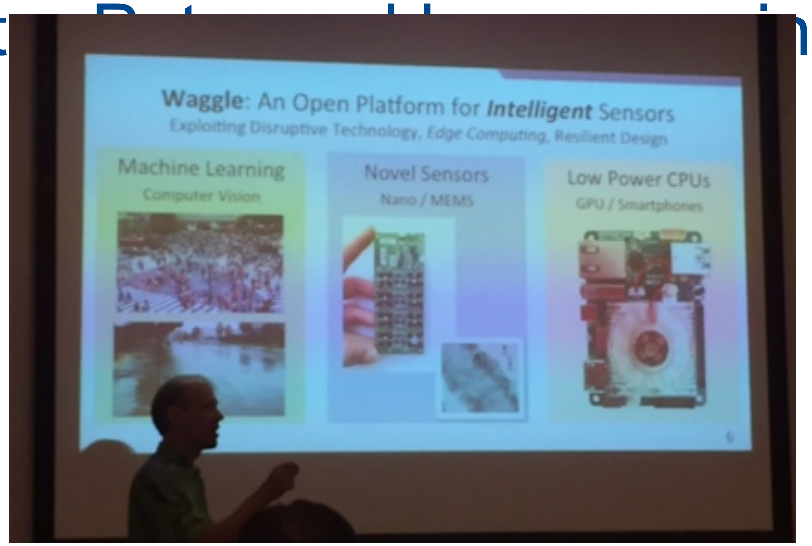
Rosa M Badia
Barcelona Supercomputing Center

EXCELENCIA SEVERO OCHOA

CCDSC 2016, La Maison des Contes, 3-6 October 2016

Challenges for this talk at CCDSC 2016

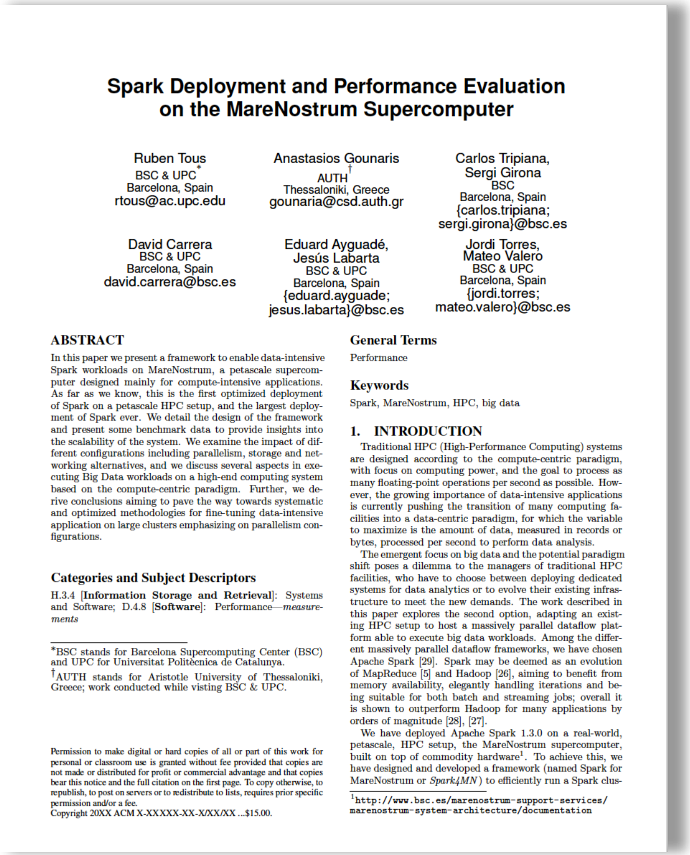
- Challenge #1: how to “uncan” my talk to meet the expectations of the workshop
- Challenge #2: how to make an interesting talk in the morning … after the first visit to the cave
- Challenge #3: how to speak aft



nterest

Goal of the presentation

- Why we do not compare Spark to PyCOMPSs?



Spark Deployment and Performance Evaluation on the MareNostrum Supercomputer

Ruben Tous
BSC & UPC[*]
Barcelona, Spain
rtous@ac.upc.edu

Anastasios Gounaris
AUTH[†]
Thessaloniki, Greece
gounaria@csd.auth.gr

Carlos Tripiana,
Sergi Girona
BSC
Barcelona, Spain
{carlos.tripiana;
sergi.girona}@bsc.es

David Carrera
BSC & UPC
Barcelona, Spain
david.carrera@bsc.es

Eduard Ayguadé,
Jesús Labarta
BSC & UPC
Barcelona, Spain
{eduard.ayguade;
jesus.labarta}@bsc.es

Jordi Torres,
Mateo Valero
BSC & UPC
Barcelona, Spain
{jordi.torres;
mateo.valero}@bsc.es

ABSTRACT

In this paper we present a framework to enable data-intensive Spark workloads on MareNostrum, a petascale supercomputer designed mainly for compute-intensive applications. As far as we know, this is the first optimized deployment of Spark on a petascale HPC setup, and the largest deployment of Spark ever. We detail the design of the framework and present some benchmark data to provide insights into the scalability of the system. We examine the impact of different configurations including parallelism, storage and networking alternatives, and we discuss several aspects in executing Big Data workloads on a high-end computing system based on the compute-centric paradigm. Further, we derive conclusions aiming to pave the way towards systematic and optimized methodologies for fine-tuning data-intensive application on large clusters emphasizing on parallelism configurations.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software; D.4.8 [**Software**]: Performance—*measurements*

*BSC stands for Barcelona Supercomputing Center (BSC) and UPC for Universitat Politècnica de Catalunya.

†AUTH stands for Aristotle University of Thessaloniki, Greece; work conducted while visting BSC & UPC.

General Terms

Performance

Keywords

Spark, MareNostrum, HPC, big data

1. INTRODUCTION

Traditional HPC (High-Performance Computing) systems are designed according to the compute-centric paradigm, with focus on computing power, and the goal to process as many floating-point operations per second as possible. However, the growing importance of data-intensive applications is currently pushing the transition of many computing facilities into a data-centric paradigm, for which the variable to maximize is the amount of data, measured in records or bytes, processed per second to perform data analysis.

The emergent focus on big data and the potential paradigm shift poses a dilemma to the managers of traditional HPC facilities, who have to choose between deploying dedicated systems for data analytics or to evolve their existing infrastructure to meet the new demands. The work described in this paper explores the second option, adapting an existing HPC setup to host a massively parallel dataflow platform able to execute big data workloads. Among the different massively parallel dataflow frameworks, we have chosen Apache Spark [29]. Spark may be deemed as an evolution of MapReduce [5] and Hadoop [26], aiming to benefit from memory availability, elegantly handling iterations and being suitable for both batch and streaming jobs; overall it is shown to outperform Hadoop for many applications by orders of magnitude [28], [27].

We have deployed Apache Spark 1.3.0 on a real-world, petascale, HPC setup, the MareNostrum supercomputer, built on top of commodity hardware[1]. To achieve this, we have designed and developed a framework (named Spark for MareNostrum or *Spark4MN*) to efficiently run a Spark clus-

[1] http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture/documentation

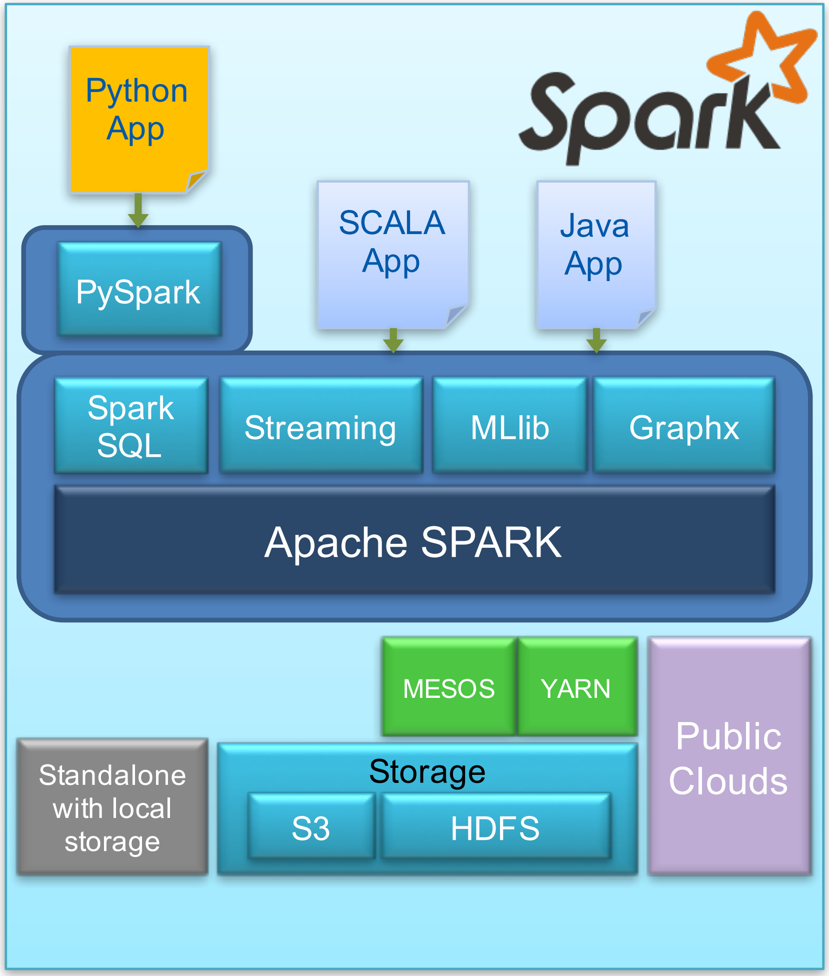
Outline

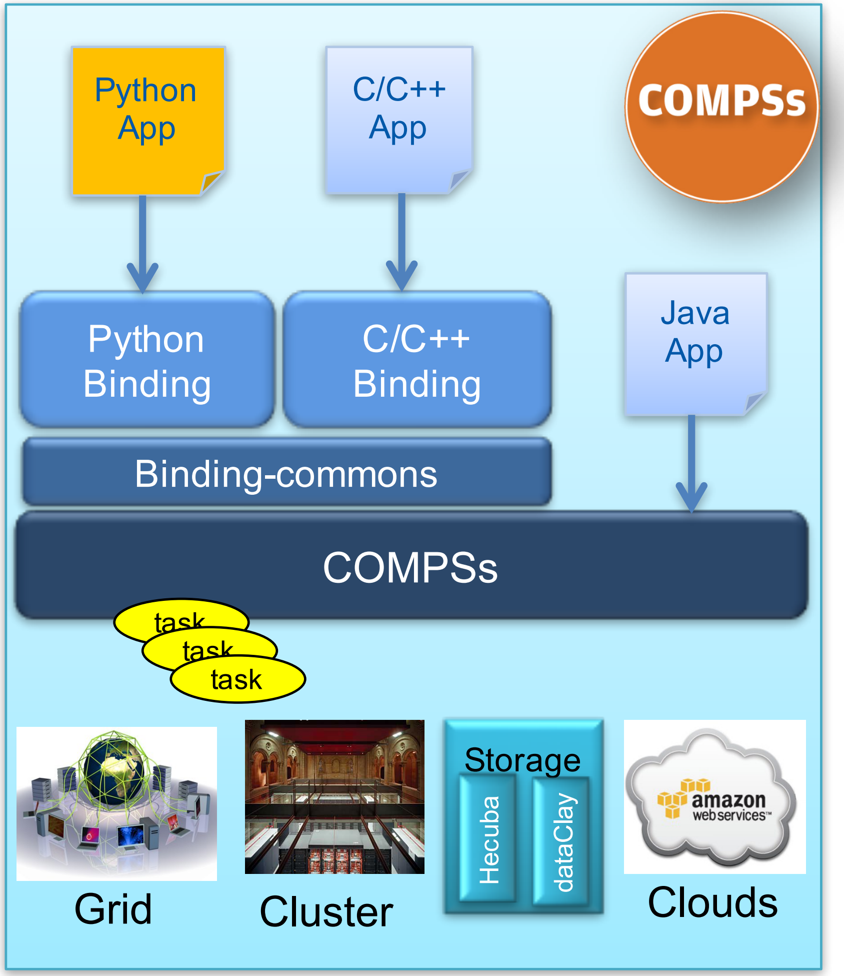
- COMPSs vs Spark
  - Architecture
  - Programming
  - Runtime
  - MN deployment
- Codes and results
  - Examples: Wordcount, Kmeans, Terasort
  - Programming differences
  - Some performance numbers
- Conclusions

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

COMPSS VS SPARK

Architecture comparison

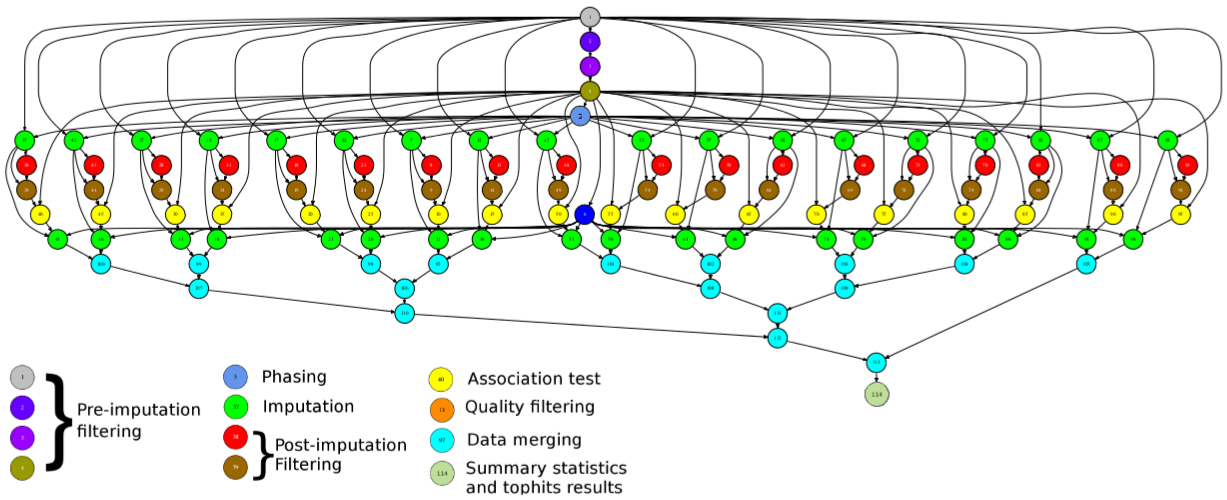




Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming with PyCOMPSs/COMPSs

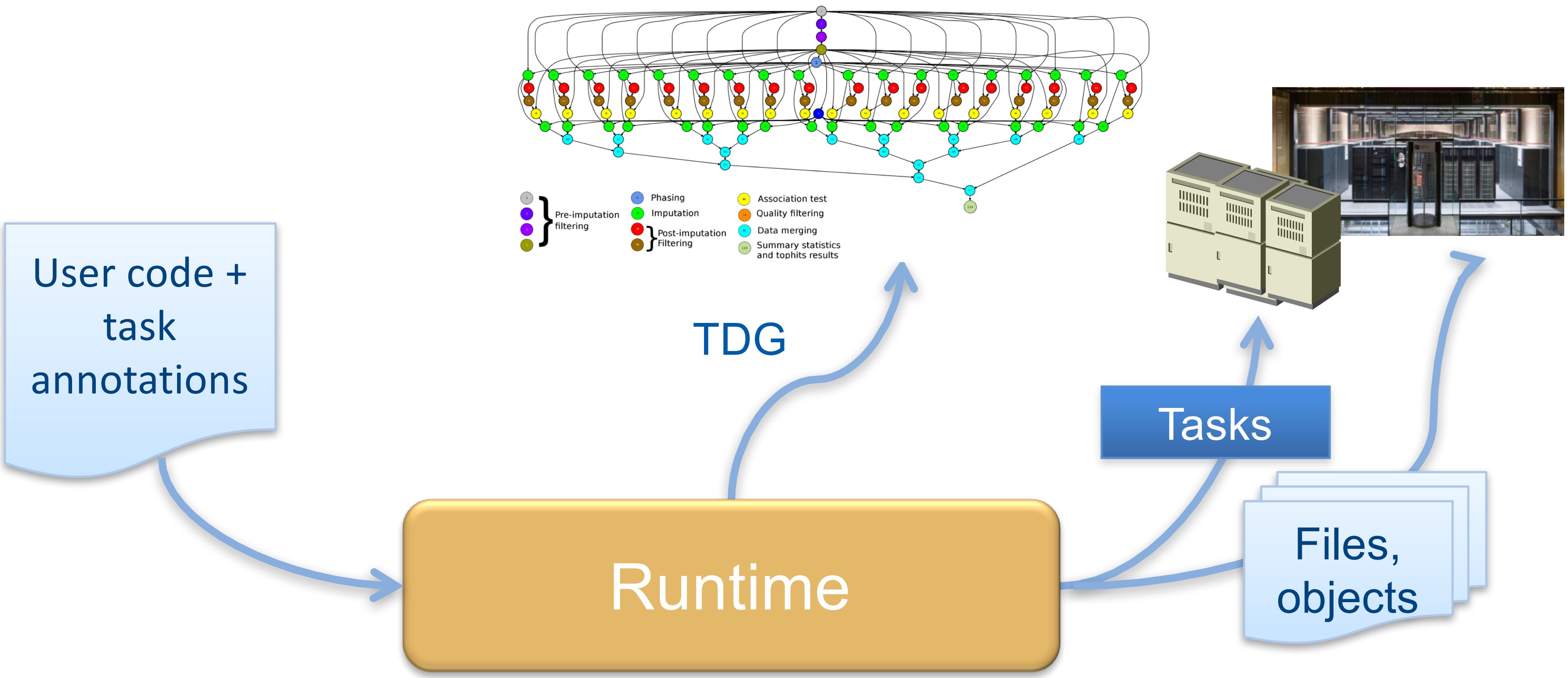
- Sequential programming
- General purpose programming language + annotations/hints
  - To identify tasks and directionality of data
- Task based: task is the unit of work
- Simple linear address space
- Builds a task graph at runtime that express potential concurrency
  - Implicit workflow
- Exploitation of parallelism
  - … and of distant parallelism
- Agnostic of computing platform
  - Enabled by the runtime for clusters, clouds and grids
  - Cloud federation



Programming with Spark

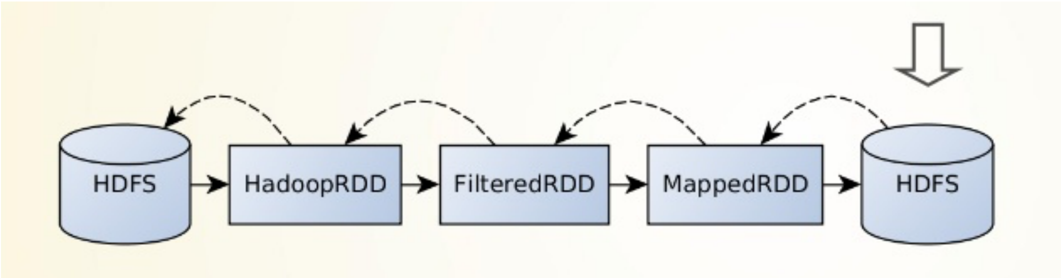
- Sequential programming
- General purpose programming language + operators
- Main abstraction: Resilient Distributed Dataset (RDD)
  - Collection of read-only elements partitioned across the nodes of the cluster that can be operated on in parallel
- Operators transform RDDs
  - Transformations
  - Actions
- Simple linear address space
- Builds a DAG of operators applied to the RDDs
- Somehow agnostic of computing platform
  - Enabled by the runtime for clusters and clouds

COMPSs Runtime behavior



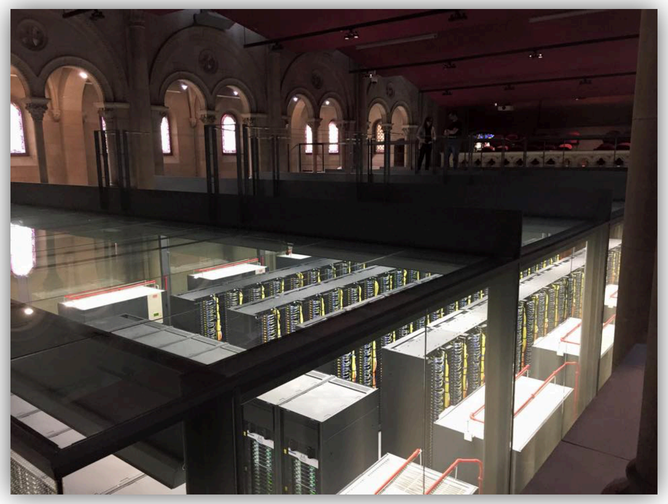
Spark runtime

- Runtime generates a DAG derived from the transformations and actions
- RDD is partitioned in chunks and each transformation/action will be applied to each chunk
  – Chunks mapped in different workers – possibility of replication
  – Tasks scheduled where the data resides
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  – Less suitable for applications that make asynchronous fine-grained updates to shared state
- Intermediate RDD can persist in-memory
- Lazy execution:
  – Actions trigger the execution of a pipeline of transformations



COMPSs @ MN

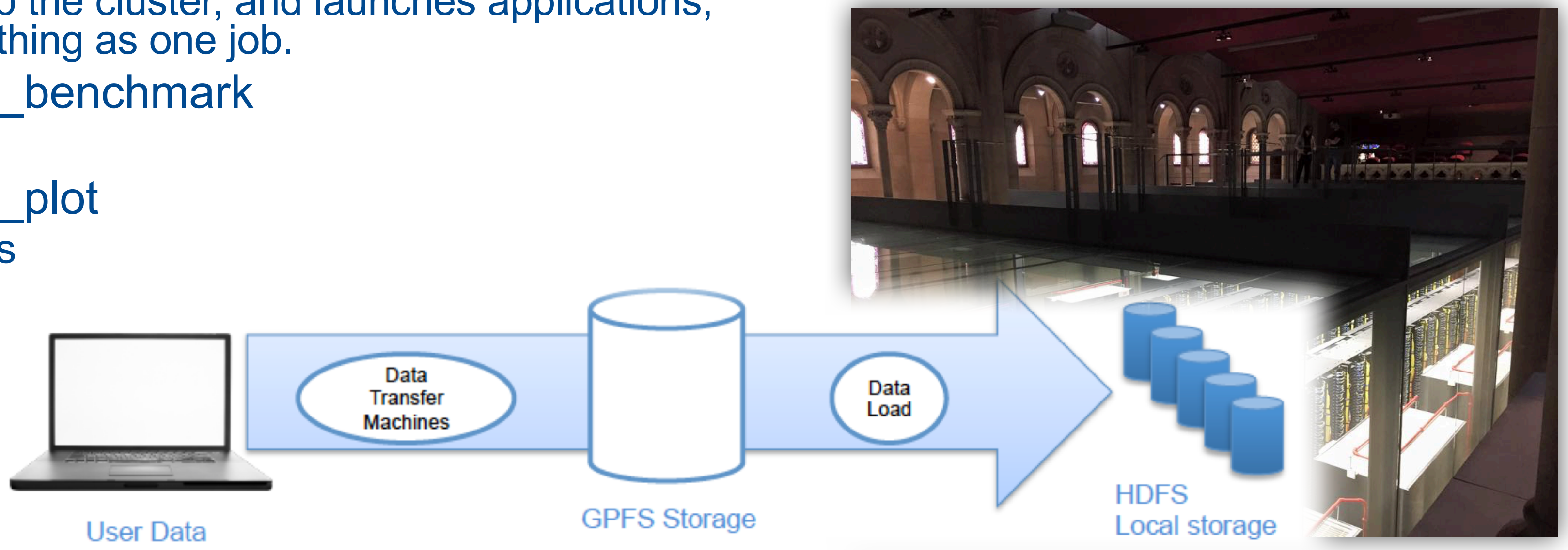
- MareNostrum version
  - Specific script to generate LSF scripts and submit them to the scheduler: enqueue_compss
  - N+1 MareNostrum nodes are allocated
  - One node runs the runtime, N nodes run worker processes
    - Each worker process can execute up to 16 simultaneous tasks
  - Files in GPFS
    - No data transfers
    - Temporal files created in local disks
- Results from COMPSs release 2.0 beta
  - To be released at SC16



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

SPARK @ MN - spark4mn

- Spark deployed in MareNostrum supercomputer
- Spark jobs are deployed as LSF jobs
  - HDFS mapped in GPFS storage
  - Spark runs in the allocation
- Set of commands and templates
  - Spark4mn
    - sets up the cluster, and launches applications, everything as one job.
  - spark4mn_benchmark
    - N jobs
  - spark4mn_plot
    - metrics



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CODES AND RESULTS

Codes

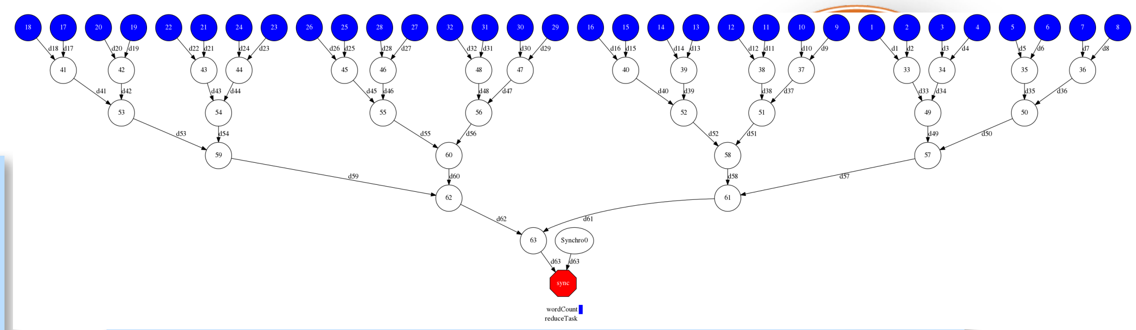
- Three examples from Big Data workloads
  - Wordcount
  - K-means
  - Terasort
- Programming language
  - Scala for Spark
  - Java for COMPSs
  - … since Python was not available in the MN Spark installation

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Code comparison – WordCount (Scala/Java)

Spark

```java
JavaRDD<String> file = sc.textFile(inputDirPath+"/*.txt");
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String,
String>() {
    public Iterable<String> call(String s) {
                        return Arrays.asList(s.split(" "));
    }
});
JavaPairRDD<String, Integer>
   pairs = words.mapToPair(new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) {
                        return new Tuple2<String, Integer>(s, 1);
    }
});
JavaPairRDD<String, Integer>
     counts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>()
{
    public Integer call(Integer a, Integer b) {
                        return a + b;
    }
});
counts.saveAsTextFile(outputDirPath);
```



```java
int neighbor=1;
while (neighbor<l){
    for (int result=0; result<l; result+=2*neighbor){
        if (result+neighbor < l){
            partialResult[result] = reduceTask (partialResult[result],
                                                partialResult[result+neighbor]);
        }
    }
    neighbor*=2;
}
int elems = saveAsFile(partialResult[0]);
```

```java
public interface WordcountItf {
@Method (declaringClass = "wordcount.multipleFilesNTimesFine.Wordcount")
public HashMap<String, Integer> reduceTask(
    @Parameter HashMap<String, Integer> m1,
    @Parameter HashMap<String, Integer> m2 );
@Method (declaringClass = "wordcount.multipleFilesNTimesFine.Wordcount")
public HashMap<String, Integer> wordCount(
    @Parameter (type = Type.FILE, direction = Direction.IN) String filePath );}
```

Code comparison – WordCount (Python)

Spark

```
from __future__ import print_function
import sys
from operator import add
from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    sc = SparkContext(appName="PythonWordCount")

    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' ')) \
              .map(lambda x: (x, 1)) \
              .reduceByKey(add)
    output = counts.collect()

    for (word, count) in output:
        print("%s: %i" % (word, count))

    sc.stop()
```

COMPSs

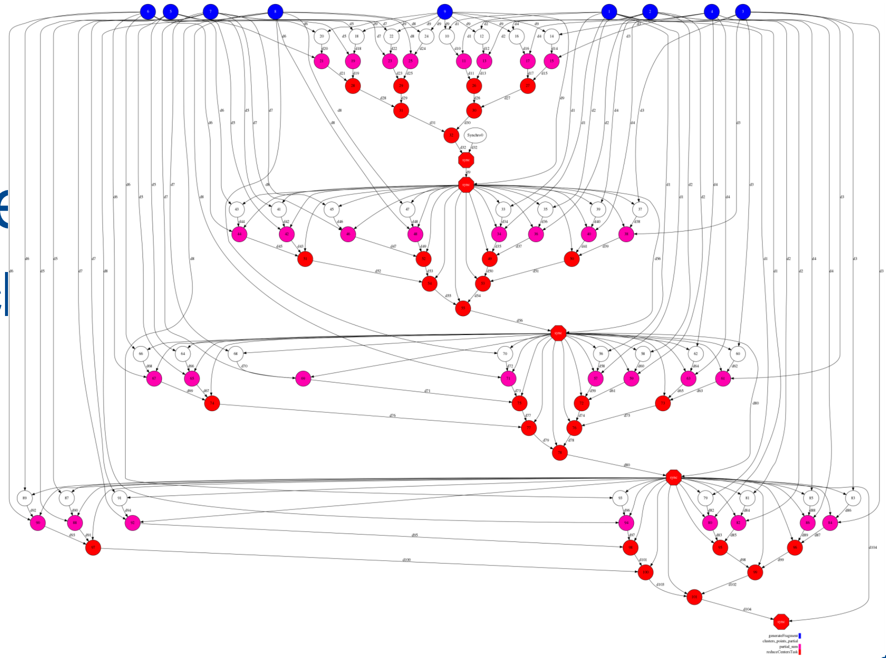
```
from collections import defaultdict
import sys

if __name__ == "__main__":
    from pycompss.api.api import compss_wait_on
    pathFile = sys.argv[1]
    sizeBlock = int(sys.argv[2])

    result=defaultdict(int)
    for block in read_file_by_block(pathFile, sizeBlock):
        presult = word_count(block)
        reduce_count(result, presult)

    output = compss_wait_on(result)
    for (word, count) in output:
        print("%s: %i" % (word, count))
```

```
@task(dict_1=INOUT)
def reduce_count(dict_1, dict_2):
    for k, v in dict_2.iteritems():
        dict_1[k] += v
```

```
@task(returns=dict)
def word_count(collection):
    result = defaultdict(int)
    for word in collection:
        result[word] += 1
    return result
```

Kmeans – code structure

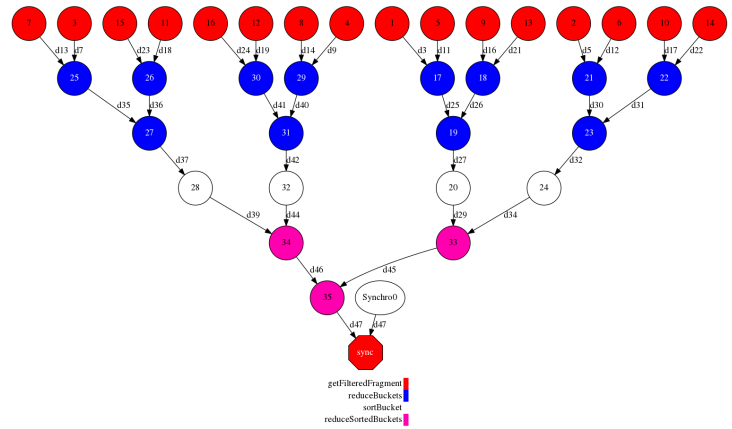
- Algorithm based on the Kmeans scala code available at MLlib
- COMPSs code written in Java, following same structure
- Input: N points x M dimensions, to be clustered in K centers
  - Randomly generated
  - Split in fragments
- Iterative process until convergence
  - For each fragment: Assign points to cl
  - Compute new centers



Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Terasort

- Algorithm based on the Terasort scala code available at github by Ewan Higgs
- COMPSs code written in Java, following same structure
- Data partitioned in fragments
- Points in a range are filtered from each fragment
- All the points in a range are then sorted



Code comparison

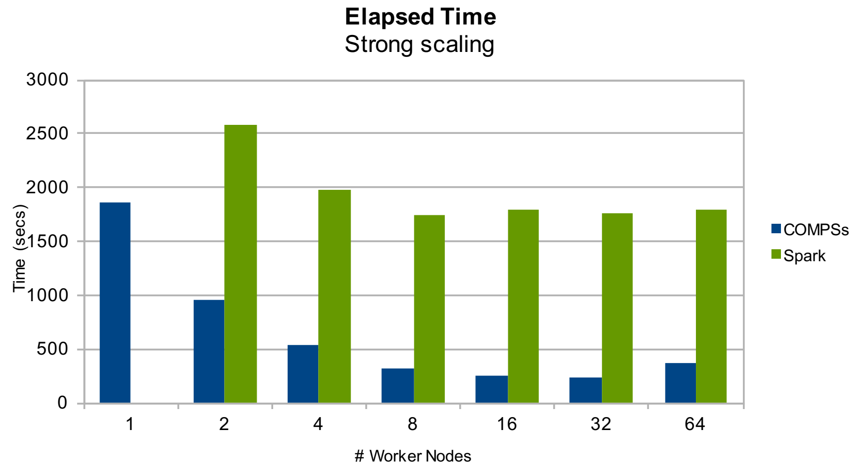
	WordCount		Kmeans		Terasort	
	COMPSs	Spark	COMPSs	Spark	COMPSs	Spark
Total #lines	152	46	538	871	542	259
#lines tasks	35		56		44	
#lines interface	20		35		34	
#tasks / #operators	2	5	4	12	4	4

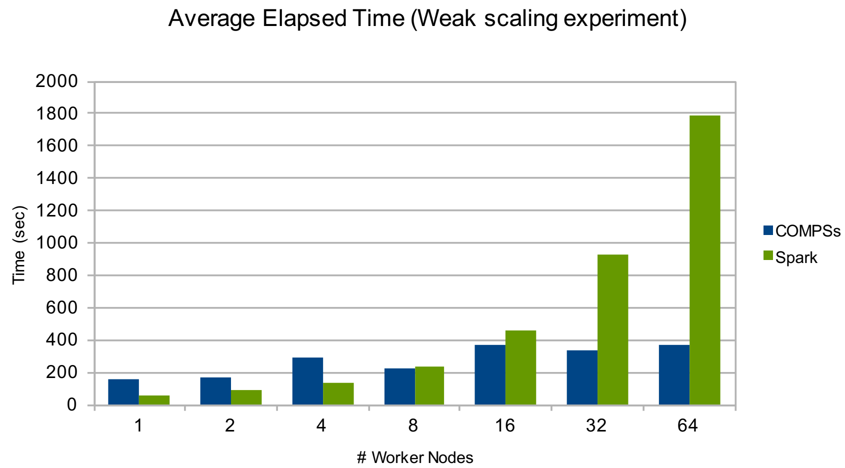
- Spark codes more compact
- Less flexible interface

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

WordCount performance

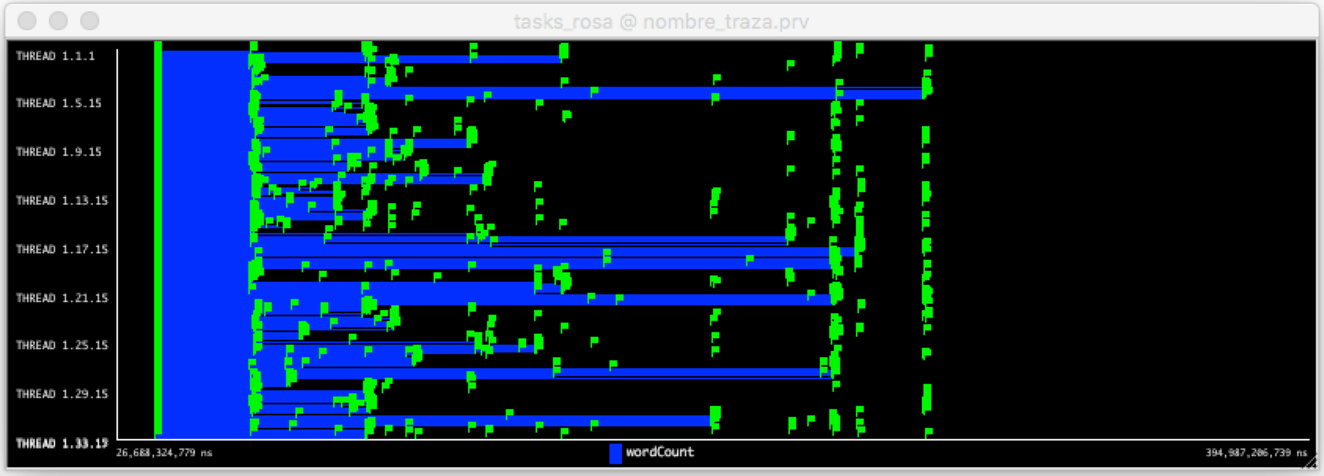
- Strong scaling
  - 1024 files / 1GB each = 1TB
  - Each worker node runs up to 16 tasks in parallel
- Weak scaling
  - 1 GB / task

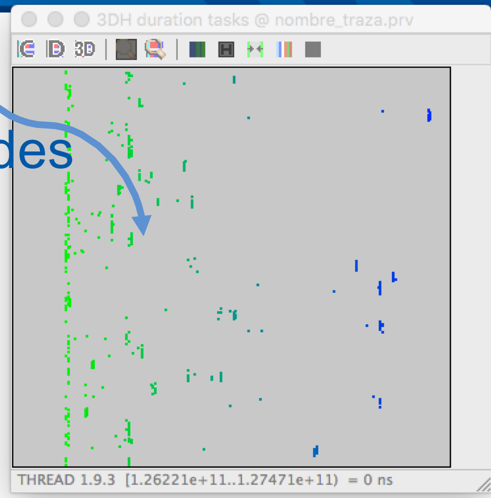




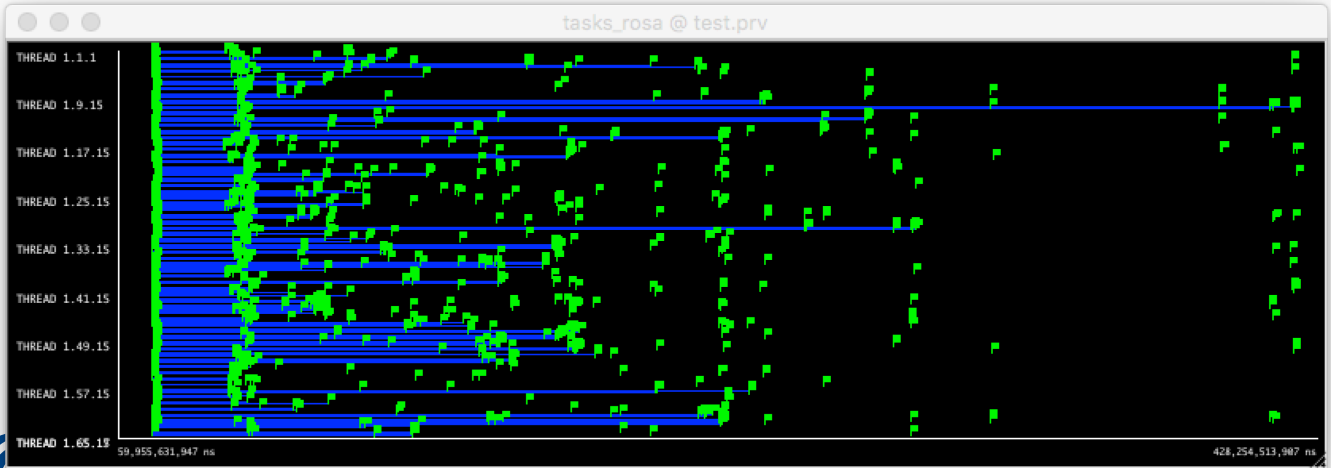
WordCount traces - strong scaling

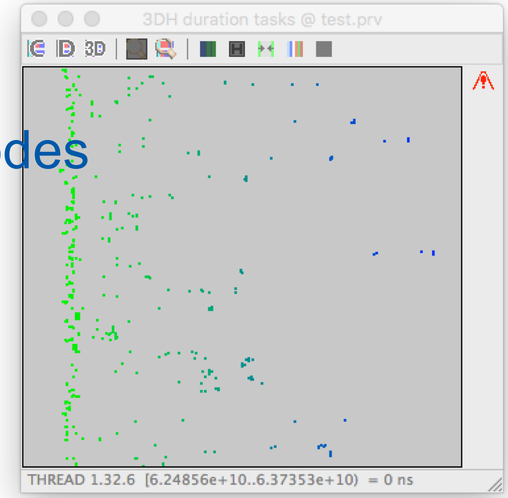
Large variability due to reads to gpfs





32 nodes



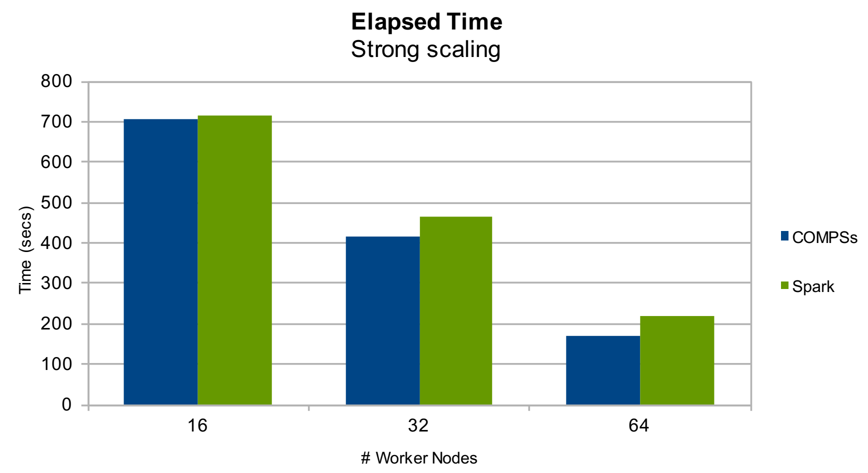


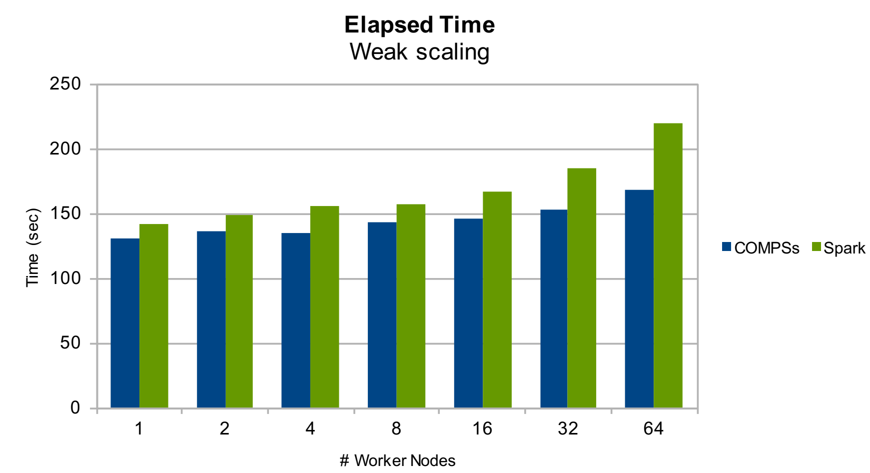
64 nodes

BSC Supercomputing Center
Centro Nacional de Supercomputación

Kmeans performance

- Strong scaling – total dataset:
  - Points 131.072,000
  - Dimensions 100
  - Centers 1000
  - Iterations 10
  - Fragments 1024
  - Total dataset size: ~100 GB
- Weak Scaling – dataset per worker:
  - Points 2.048,000
  - Dimensions 100
  - Centers 1000
  - Iterations 10
  - Fragments 16
  - Dataset size: ~1.5 GB

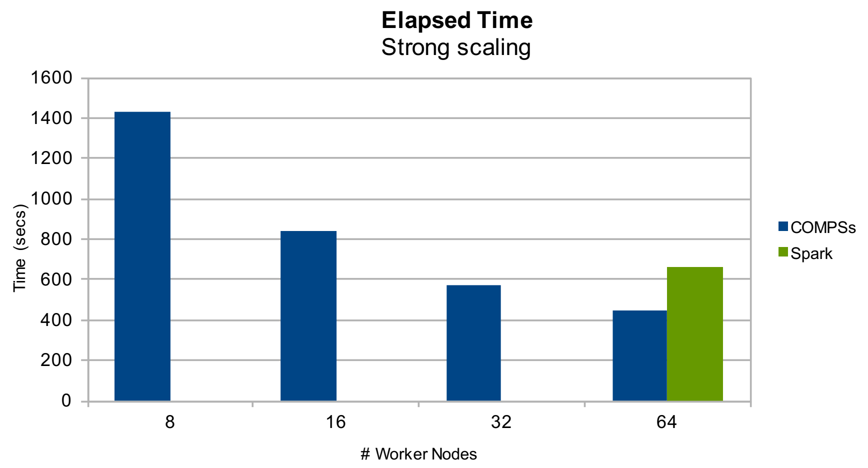


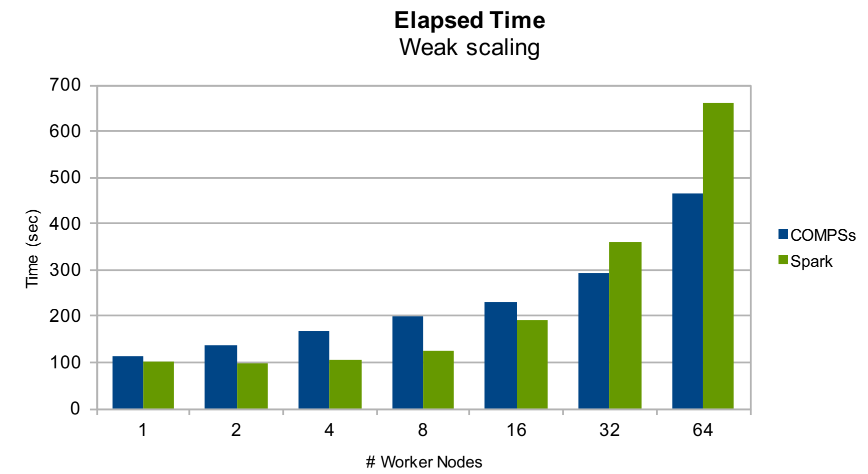


Terasort performance

- Strong Scaling
  - 256 files / 1 GB each
  - Total size 256 GB

- Weak scaling
  - 4 files / 1 GB per worker
  - 4 GB / worker

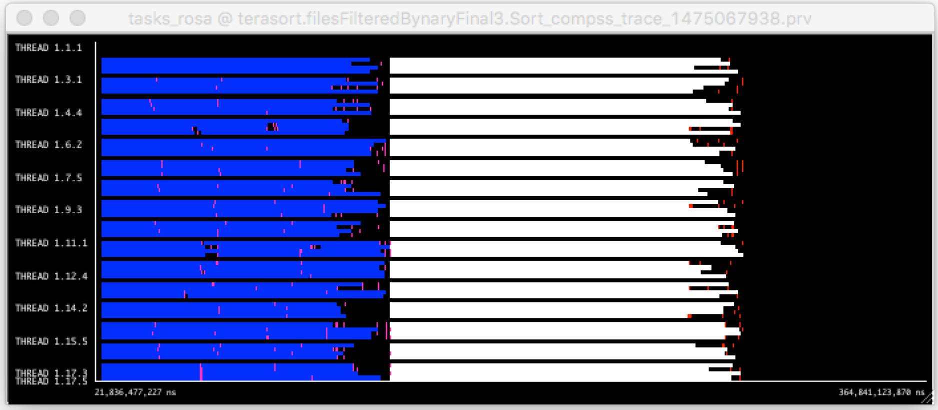


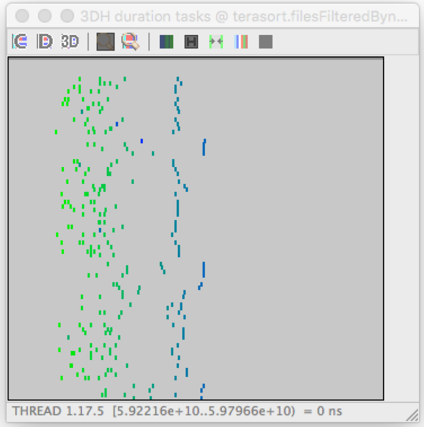


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

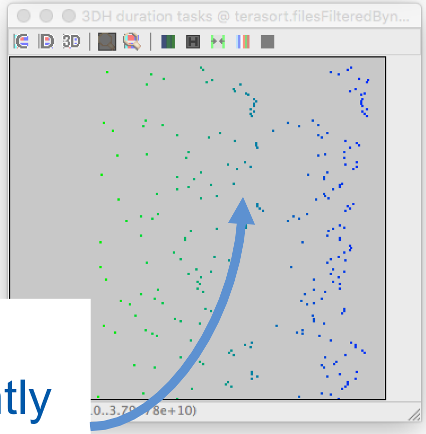
Terasort traces – weak scaling

16 nodes





32 nodes



Sort task duration increases significantly
+ large variability
Reads/writes from file

Conclusions

- Summary of comparison
  - Spark code is more compact
  - COMPSs offers more flexibility, both in programming model and runtime behavior
  - Performance results slightly better for COMPSs
  - Need to better understand reasons for better performance
- Ongoing work:
  - Integration with new storage technologies:
    - dataClay, Hecuba
    - Will improve current issues with traditional file systems (gpfs)
  - Support to end-to-end HPC workflows
    - COMPSs runtime enabled to run MPI workloads as tasks
    - Support for streaming
- Future plans
  - Promotion of PyCOMPSs in Python community
    - Enablement of automatic installation (pip install)
- Distribution
  - compss.bsc.es

Maybe we will not kill the giant…

…but we will try hard


www.bsc.es
BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
Thank you!