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

Rosa M Badia
Barcelona Supercomputing Center

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 after Pete and keep your interest
Goal of the presentation

Why we do not compare Spark to PyCOMPSs?
Outline

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

Apache SPARK

- Python App
- SCALA App
- Java App
- PySpark
- Spark SQL
- Streaming
- MLlib
- Graphx
- MESOS
- YARN
- Storage
- S3
- HDFS
- Public Clouds
- Standalone with local storage

COMPSs

- Python App
- C/C++ App
- Python Binding
- C/C++ Binding
- Binding-commons
- Java App
- Grid
- Cluster
- Clouds
- Storage
- Hecuba
- DataClay

Python
SCALA
Java
Python
C/C++
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

User code + task annotations

Runtime

TDG

Tasks

Files, objects
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
**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
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
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
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);

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]);

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 );
}
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()

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 closest center
  - Compute new centers
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

<table>
<thead>
<tr>
<th></th>
<th>WordCount</th>
<th></th>
<th></th>
<th>Terasort</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COMPSs</td>
<td>Spark</td>
<td>COMPSs</td>
<td>Spark</td>
<td>COMPSs</td>
<td>Spark</td>
</tr>
<tr>
<td>Total #lines</td>
<td>152</td>
<td>46</td>
<td>538</td>
<td>871</td>
<td>542</td>
<td>259</td>
</tr>
<tr>
<td>#lines tasks</td>
<td>35</td>
<td>56</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines interface</td>
<td>20</td>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#tasks / #operators</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Spark codes more compact

Less flexible interface
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
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
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
Thank you!