Università degli studi di Firenze
Scuola di Ingegneria

Histograms Representation of Name Entity Detector

Elena Sesoldi, Giulia Pellegrini
parallel18computing@gmail.com

Febrary 26, 2018

Content

- Implementation of an algorithm that extraxts Named Entities from tweet messages and computes for each different category of considered entity the corresponding histogram of occurences

- Implementation of an algorithm that extraxts Named Entities from tweet messages and computes for each different category of considered entity the corresponding histogram of occurences
- 2 programming languages: C++ & Java

Introduction
Task

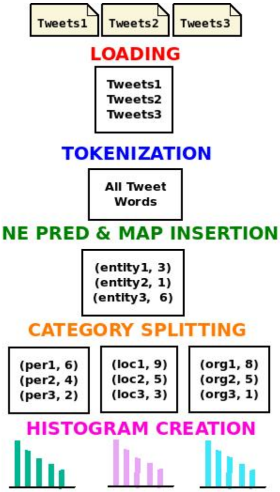
- Implementation of an algorithm that extraxts Named Entities from tweet messages and computes for each different category of considered entity the corresponding histogram of occurences
- 2 programming languages: C++ & Java
- A sequential implementation: C++

- Implementation of an algorithm that extraxts Named Entities from tweet messages and computes for each different category of considered entity the corresponding histogram of occurences
- 2 programming languages: C++ & Java
- A sequential implementation: C++
- 2 parallel implementations: OpenMP & Hadoop

C++ Implementation
Sequential

Steps:

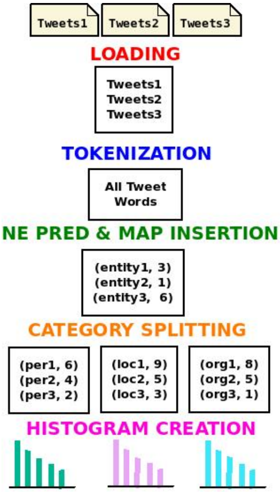
- dataset loading
- tokenization
- Named Entity prediction & map insertion (for loop)
- category splitting
- histograms' creation



C++ Implementation
Sequential to OpenMP

Steps:

- dataset loading
- tokenization
- Named Entity prediction & map insertion (for loop)
- category splitting
- histograms' creation



Parallel loop

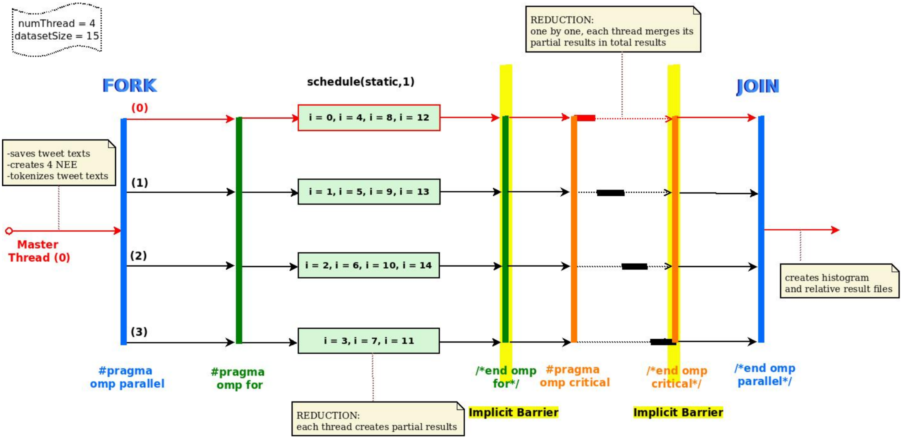
After creating the tokens as in the sequential version, the parallel part of the program starts with the directive ♯ ***pragma omp parallel***. Then with the directive ♯ ***pragma omp for*** a for loop is executed in parallel and distributed among threads already existing in the parallel region.

Reduction

In the loop, a partial result map is defined and created for each thread. After the loop it is defined a critical section with the clause ♯ ***pragma omp critical*** where each thread merges its partial result in a final common one.

C++ Implementation

OpenMp API



for loop scheduling

The schedule clause avails to control the distribution of the loop between threads.

- ***schedule(static, 1)***
  In this case it divides the cycle into pre-defined blocks of given size (one) and the blocks are statically assigned to the threads sequentially following the identifier of the individual thread. Once the available threads are finished, the blocks are reassigned from the first thread.

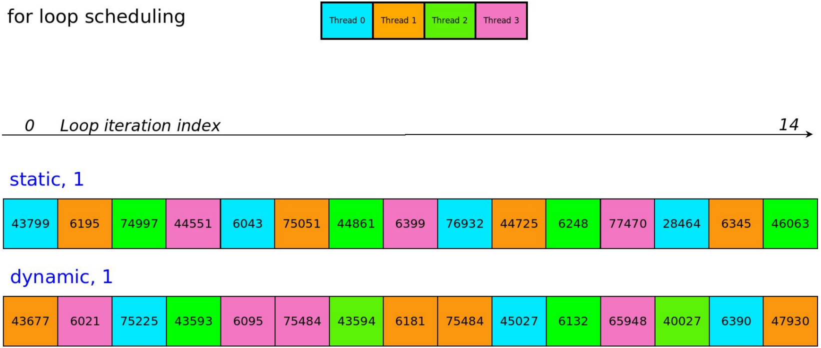
for loop scheduling

The schedule clause avails to control the distribution of the loop between threads.

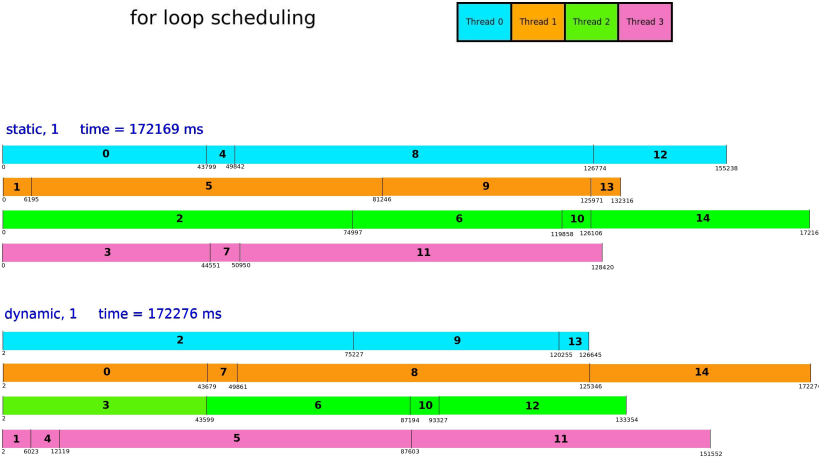
- ***schedule(static, 1)***
  In this case it divides the cycle into pre-defined blocks of given size (one) and the blocks are statically assigned to the threads sequentially following the identifier of the individual thread. Once the available threads are finished, the blocks are reassigned from the first thread.
- ***schedule(dynamic, 1)***
  A block is assigned to the first available thread to execute it. At the end of its execution that same thread can be used again to calculate another block.

C++ Implementation
OpenMp Schedule

for loop scheduling



Implementation
OpenMp Schedule



We test our program with the same three datasets, repeating them several times, through setting a variabile named *numRepetition*. In each execution we use a number of datasets equal to three multiplied by *numRepetition*.

We tried more parallel versions with different number of threads using *omp_set_num_threads(int num_threads)*, to see how execution times changes.

Rep	Seq	Omp1	Omp2	Omp3	Omp4
1	66533	66727	63783	44663	44529
2	133256	134112	72604	84968	66612
3	199776	200994	135204	127669	100748
4	266349	267882	147127	142258	123365
5	336083	334772	211378	184322	164915

Once the execution times have been detected we compute the relative **Speedup**

Rep	Sp1	Sp2	Sp3	Sp4
1	0.9970	1.0431	1.4896	1.4941
2	0.9936	1.8353	1.5683	2.0004
3	0.9939	1.4775	1.5647	1.9829
4	0.9942	1.8103	1.8722	2.1598
5	1.0039	1.5899	1.9861	2.2198

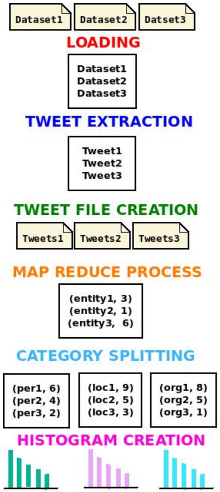
Once the execution times have been detected we compute the relative **Speedup**

Rep	Sp1	Sp2	Sp3	Sp4
1	0.9970	1.0431	1.4896	1.4941
2	0.9936	1.8353	1.5683	2.0004
3	0.9939	1.4775	1.5647	1.9829
4	0.9942	1.8103	1.8722	2.1598
5	1.0039	1.5899	1.9861	2.2198

Java Implementation
Java + MapReduce

Steps:

- dataset loading
- tweet texts' extraction
- tweet files' creation
- Map Reduce Process
- category splitting
- histograms' creation



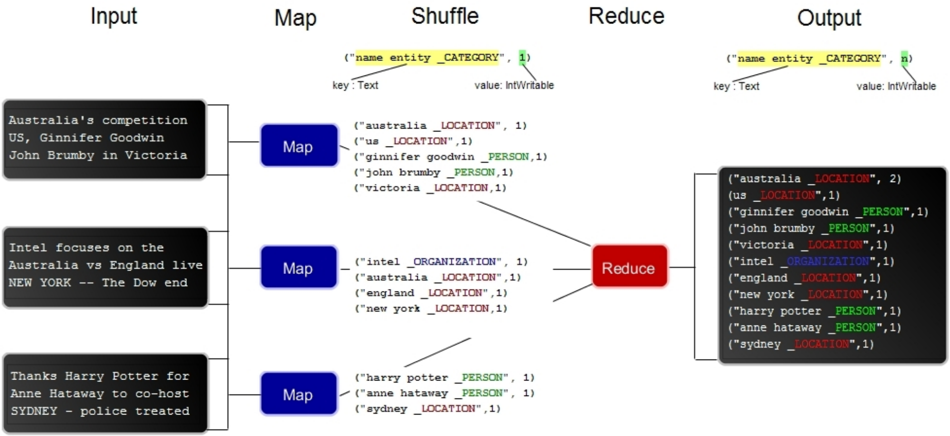
MapReduce

The task of the *MapReduce* framework is counting the number of named entities in a collection of *.xml* files containg tweet messages.

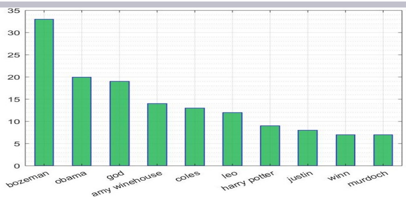
- ***Mapper***
  Processes line by line the input files, looking for the entities. Once obtained the chunk set of entities, it loops on it creating the key-value pairs. The key is a string of the entity combined with its category and the value is an integer always setting at one.
- ***Reducer***
  Takes all the key-value pairs for a given real-word object and adds up the values, generating a single key-value pair for each entity, that is written on the context.

Java Implementation

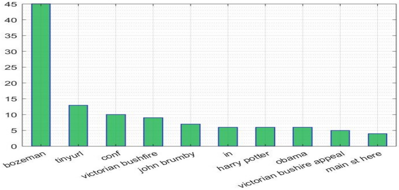
Apache Hadoop Framework



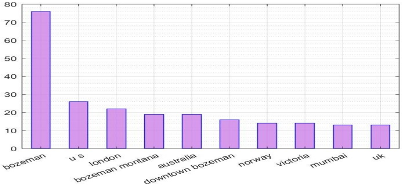
Histograms



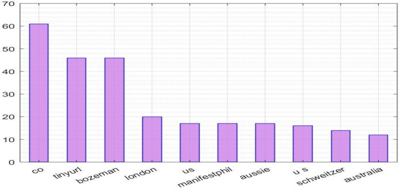
35
30
25
20
15
10
5
0
bozeman
obama
god
amy winehouse
coles
leo
harry potter
justin
winn
murdoch



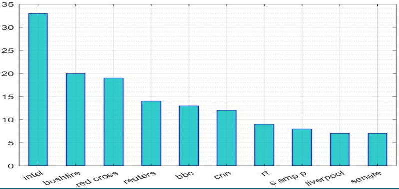
45
40
35
30
25
20
15
10
5
0
bozeman
tinyurl
conf
victorian bushfire
john brumby
in
harry potter
obama
victorian bushire appeal
main st here



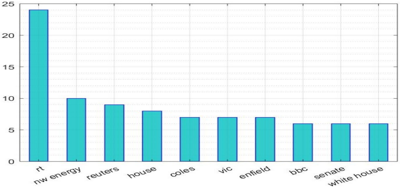
80
70
60
50
40
30
20
10
0
bozeman
u s
london
bozeman montana
australia
downtown bozeman
norway
victoria
mumbai
uk



70
60
50
40
30
20
10
0
co
tinyurl
bozeman
london
us
manifestphil
aussie
u s
schweitzer
australia



35
30
25
20
15
10
5
0
intel
bushfire
red cross
reuters
bbc
cnn
rt
s amp p
liverpool
senate



25
20
15
10
5
0
rt
nw energy
reuters
house
coles
vic
enfield
bbc
senate
white house

Conclusions & Applications

Conclusions

- three different implementations of the creation of histograms that represent the frequence of named entity in tweets.
- two version written in C++, sequential and with the shared memory OpenMP API, and one in Java with the distributed memory Apache Hadoop tool.
- for the extraction of named entities we benefit of a library for each programming language; the histograms are different because theese have distinct NER methodologies, although both are based on Machine Learning.
- we test our programs to see the benefits of the parallelization. For C++ we can evaluate the parallelism computing the speed up, and we obtained the best value equals to 2.2198 with four threads and fifteen total datasets

Conclusions & Applications

Aplications

- Information Extraction
- Marketing Initiatives
- Information Security

Thank you for the attention