Ref. Ares(2018)6678825 - 31/12/2018

E2Data

D2.2.a

Intermediate Applications' Definitions (report)

Neurocom Luxembourg SA

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

D2.2.a

Intermediate Applications' Definitions (report)

DOCUMENT IDENTIFIER:	D2.2
DUE DATE:	31/12/2018
DELIVERY DATE:	
CLASSIFICATION:	RESTRICTED
EDITORS:	Vassilis SPITADAKIS, Ioannis Komnios
DOCUMENT VERSION:	2.0

CONTRACT START DATE:	**1st January 2018**
CONTRACT DURATION:	36 months

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

CONTRIBUTORS	
Name	**Organization**
V. Spitadakis	Neurocom Luxembourg SA
C. Tsalidis	Neurocom Luxembourg SA
F.Hernu	Neurocom Luxembourg SA
O. Akrivopoulos	Spark Works ITC
N. Kanakis	Spark Works ITC
M. Kritikos	EXUS
L. Lefkokoilos	EXUS
G. Tselios	EXUS
G. Mylonas	CTI
D. Strimpel	iProov

PEER **REVIEWERS**	
Name	**Organization**
N. Kanakis	Spark Works ITC
D. Strimpel	iProov

REVISION HISTORY

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Version	Date	Modifications
0.1	8/10/2018	Outline
0.2	20/11/2018	Integrating Use Cases feedback - Security use case
0.3	20/11/2018	Integrating Use Cases feedback - Green Buildings use case
0.4	10/12/2018	Integrating Use Cases feedback - Financial (NLP) use case
0.5	18/12/2018	Integrating Use Cases feedback - Health Use case
1.0	21/12/2018	Final integration – 1st complete draft
1.5	27/12/2018	Peer Review
2.0	30/12/2018	Revision made – Final draft

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Executive Summary

All use cases have commenced their tasks towards accelerating critical code parts based on E2data. Progress has been mainly done in the direction of developing the accelerated version, wherever that has been possible, based on the current version of the E2Data Heterogeneous execution engine, i.e. Tornado.

The health use case implementation for batch ALS algorithm has already completed. The Apache Flink implementation is ongoing. The Fintech use case (NLP based) has already completed the WDK kernel, now adapted to the E2Data Tornado version. This will be a driver for the rest of the development, before the improved kernels become integrated into the NLP application. For the Green Building Use Case, the code of the kernels was modified to meet the requirements of the current status of the E2Data execution engine. Based on this, the kernels have been adapted to a batch processing version. Apart from the already ported to Apache Flink batch processing version, a stream analytics version will be developed within next stages. For the biometric recognition use case, work is ongoing. Benchmark applications are developed while E2data tornado platform needs to enhance its support on specific processing in order to better respond to the use case requirements.

The preliminary tests of accelerated code kernels, do provide hints for E2data team, to investigate further specific functionalities, challenging adjustment of tornado to achieve improved performance of use case kernels. Continuous work with iterations and interactions between use case adapters, testers and core E2Data development team, will drive the remaining work. All use cases need to improve their performance through E2Data, while progressing towards completing the overall envisaged functionality of the modified kernels in the respective use cases. Remaining implementation activities include: Benchmarking on multiple load scenarios, integration with Flink, acceleration of remaining kernels, revisiting implementations with newer Tornado versions.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Contents

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Introduction

E2Data's vision is to develop a novel Big Data software stack that will help Big Data practitioners to exploit in a transparent and efficient manner the available underlying diverse heterogeneous hardware resources without requiring software re-engineering by the programmers. While in the de-facto scale-out or homogeneous scale-up model the applications are partitioned and sent for execution on CPU nodes, E2Data will intelligently identify which parts of the applications can be hardware accelerated and, **dynamically**, based on the current hardware resources will send tasks for execution on the appropriate nodes. The scheduling, compilation, and execution of the tasks will happen "on-the-fly" without requiring Big Data practitioners to write not-portable, low-level code for each specific device or accelerator. This will ultimately translate to: a) **higher performance** of Big Data execution, b) **energy efficient execution**, c) **significant cost improvements** for cloud providers and end-users, and d) **enhanced scalability and performance portability** of Big Data applications.

To achieve its ambitious goals, E2Data follows an **application-driven approach** in which the development of the proposed solutions is guided by the SLAs of the industrial partners. The use case providers presented an interesting set of code kernels that will be exercised. It includes:

- A matrix inversion algorithm;
- A matrix multiplication algorithm;
- Computation of intermediate matrices X and Y of Alternating Least Squares algorithm;
- An algorithm for lexicographical approximate matching in vocabularies using distances between words;
- Approximate Matching in dictionaries stored as Directed Acyclic Graph Words;
- Cosine similarity applied on words or q-grams and a variation of the algorithm in order to rank documents;
- Fuzzy Matching of terms in terminological dictionaries stored in Compressed Tries;
- Computing the sum / maximum / minimum / average of sensor measurements;
- An algorithm that takes an array of RGB images containing faces together with a set of feature location for the faces, and 'morphs' each individual image to a standard face which is finally returned.

The aim of this deliverable is to present the progress at M12 of all use cases regarding their plans towards accelerating the kernels involved in the above mentioned applications, algorithms and codes. The deliverable can be considered as incomplete, mainly aiming to provide a snapshot of the current status of the implementations. Its next version, will be the final one, complete at M24.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

1. Overview

All use cases have commenced their tasks towards accelerating critical code parts based on E2data. Progress per use case is being described in the following sections. A summarized view is given below.

Health Use Case - For this use case the Alternating Least Square (ALS) algorithm has been implemented in Java. Also, benchmarks for the batch implementation have been performed in order to find the points that need to accelerate. Furthermore, different size of datasets has been tested to have a better view of execution time. Moreover, an Apache Flink version of code is ongoing. MapReduce approach has been used for transforming and parallelize the high cost computation of ALS algorithm.

Fintech (Natural Language Processing) – Among the NLP kernels, WDK has been already implemented with Tornado SDK version 0.1.0 and will be used as a driver for the remaining kernels. It shares the same architecture and similar API. A number of tests have been implemented, that use different techniques in order to find the efficient ways to work with tornado and acceleration. For this, different example cases are being studied with lexicons and test data of various sizes and calling methods. The tests are scaled from simple cases with few words in lexicon and one input word to check to more complex cases with thousands words in lexicon and tenths of search words.

Green Building Infrastructure – The code of the kernels has been modified to meet the requirements of the current status of the E2Data execution engine. Based on this, the kernels have been adapted to a batch processing version while normally the Spark Works analytics engine utilizes stream processing. Based on this, the current version of the kernels accept an array of primitive double values while, according to the requirements of the analytics engine it should be ready to process a stream of composite objects.

Security and biometric authentication – A Java implementation of the ColourMorph kernel has been made available; the kernel is used in the biometric authentication pipeline. For benchmarking, random RGB input images are generated in varying sizes. The kernel takes as input a series of RGB images containing face images together with a set of landmarks for each image. It keeps a reference image with a set of landmarks and the value of each pixel in the reference image is set to that of the closest pixel in each of the source images. Preliminary benchmarks for the kernel have been performed, one for the unaccelerated serial Java implementation, and one for the hand-crafted CUDA version. The CUDA time was lower than the Java time as expected, but low-level caching is likely to introduce noise and rendering the times imprecise. The kernel source code has been delivered to the Tornado team for acceleration. The code involves the use of 3-dimensional arrays of primitives which are not currently supported by Tornado but support in future versions is planned.

The preliminary tests provide hints for E2Data team to investigate further specific functionalities, challenging adjustment of Tornado to achieve improved performance of use case kernels. Continuous work with iterations and interaction between use case providers and developers, testers and core E2Data development team will drive the remaining work. All use cases need to improve their performance through E2Data, while progressing towards and completing the overall envisaged functionality of the modified kernels in the respective use cases. Ongoing implementation plan includes:

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Benchmarking on multiple load scenarios, integration with Flink, acceleration of remaining kernels, revisiting implementations with newer Tornado versions.

2. Health Use Case

EXUS has developed a Java implementation of the Alternating Least Squares (ALS) algorithm. In conjunction with this, scripts are developed which create random testing datasets of varying sizes and handle CSV file I/O functionality. These are executed independently of the kernels.

In view of the current implemented Alternating Least Squares algorithm, we are delivering our implementation to the Tornado team for acceleration.

For the Health Use Case, the execution of the ALS Algorithm requires a Matrix (R m x n) with m being the number of patients and n being the number of medical conditions. Also, some other parameters need to be defined:

- Regularization (λ), that it used to avoid overfitting of training data;
- The number of factors (f);
- The number of iterations (num_iter) required to compute the cost function.

Our goal is to compute the matrix of patients X (m x f) and the matrix of medical conditions Y (n x f), so that we can minimize the cost function J.

The ALS algorithm steps (in a pseudocode presentation) are:

```
X→ init_array()
Y→ init_array()
for each  iteration in range(num_iter):
      (a) for each i in range(m):
                  dot1 =  Y.transpose * Y
                  (c) inv = (dot1 + λ*I).inverse
                  dot2 =  inv *  Y.transpose
                  dot3 = dot2 * Ri
                  Xi = dot3
      end:
      (b) for each j in range(n):
                  dot1 =  X.transpose * X
                (d) inv = (dot1 + λ*I).inverse
                  dot2 =  inv *  X.transpose
                        dot3 = dot2 * Rj
            Yj = dot3
      end:
end:
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

The steps that can be implemented with parallel execution and, in general, be improved in terms of performance are:

- Steps **(c) & (d)**, which present the matrix inverse function;
- All the matrix multiplication actions (**dot1**, **dot2**, **dot3**);
- Steps **(a) & (b)**, which represent the computations for each patient and each medical condition.

2.1. Code Kernel "Minv"

As seen in the ALS algorithm above, one of the functionalities of ALS Algorithm is the inverse function (c). In this step, we have to inverse a square matrix of size (f x f). This function has polynomial complexity ($O(f^3)$) by the number of factors. The steps of invert function are:

1) Transform the matrix into an upper triangle (Gaussian elimination phase);
2) Update the matrix with the ratios stored;
3) Perform backward substitutions.

```
public boolean matrixInvert(Matrix v)
{
	double[][] matrixForInverse = v.getData();
	int lengthOfArray = matrixForInverse.length;
	double[] helperForColumnAndRow = new double[lengthOfArray];
	double[] UpdatedRowForMatrix = new double[lengthOfArray];
	double[] storePivotingOrder = new double[lengthOfArray];
	double AB = 0;
}
```

The above method gets as input a matrix V and transforms it to its inverse matrix.

2.1.1. Specifying the kernel for heterogeneous execution

According to Section 2.1, the first step of matrix inversion process is to transform the input matrix into an upper triangle matrix and update the concatenate identity matrix that we want to transform as our inverse matrix. To achieve this, in this step we perform some basic row operations. This is the most time expensive task of this method. The code implementation of this process is shown below.

```
for (M = 0; M < lengthOfMatrix; M++)
{
	double Big = 0;
	for (L = 0; L < lengthOfMatrix; L++)
	{
		AB = Math.abs(matrixForInverse[L][L]);
		if ((AB > Big) && (storePivotingOrder[L] != 0))
		{
			Big = AB;
				K = L;
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
            }
      }
      if (Big == 0)
      {
            return false;
      }
      storePivotingOrder[K] = 0;
      UpdatedRowForMatrix[K] = 1 / matrixForInverse[K][K];
      helperForColumnAndRow[K] = 1;
      matrixForInverse[K][K] = 0;
      if (K != 0)
      {
            for (L = 0; L < K; L++)
            {
                  helperForColumnAndRow[L] = matrixForInverse[L][K];
                  if (storePivotingOrder[L] == 0)
                  {
                        UpdatedRowForMatrix[L] =

matrixForInverse[L][K] * UpdatedRowForMatrix[K];
                  }
                  else
                  {
                        UpdatedRowForMatrix[L] =

-matrixForInverse[L][K] * UpdatedRowForMatrix[K];
                  }
                  matrixForInverse[L][K] = 0;
            }
      }
      if ((K + 1) < lengthOfMatrix)
      {
            for (L = K + 1; L < lengthOfMatrix; L++)
            {
                  if (storePivotingOrder[L] != 0)
                  {
                        helperForColumnAndRow[L]                    =
matrixForInverse[K][L];
                  }
                  else
                  {
                        helperForColumnAndRow[L]                    =
-matrixForInverse[K][L];
                  }
                  UpdatedRowForMatrix[L] =

-matrixForInverse[K][L] * UpdatedRowForMatrix[K];
                  matrixForInverse[K][L] = 0;
            }
      }
}
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

2.1.2. Findings so far

Based on Section 2.1.1, the piece of code that is highlighted can be modified, using a distributed approach, in order to speed up the execution process. Also, as mentioned above, the computation time of this process only depends on the size of the matrix that we want to inverse. In particular, the length of this matrix is equal to the number of factors that we have defined in our parameters. actor f must be less that m and n (number of patients and number of medical condition).

2.1.3. Next Steps

Next steps for this kernel implementation include:

- Define the number of factors. We run the batch implementation several times with different number of factors, to find the optimal value that minimizes the cost function. Some results are already available, but we are pushing to further optimize the cost function.

- If this number is big enough (namely larger than 500), we need to convert the execution from sequential to parallel.

2.2. Code Kernel "Mdot"

Another functionality of ALS Algorithm is the multiplication function. This method also has polynomial complexity ($O(f^3)$) where f is the number of factors defined by the number of rows and columns of the first matrix and columns of the second matrix.

```
public Matrix dot(Matrix A, Matrix B){

	int rowsA = A.getRows();

	int colsB = B.getCols();

	Matrix C = new Matrix(rowsA,colsB);
}
```

The above method get as input matrix A and matrix B, multiplies them and returns the output matrix C.

2.2.1 Specifying the kernel for heterogeneous execution

Below is a standard implementation of matrix multiplication.

```
for (int i = 0; i < rowsA; i++) { // aRow
	for (int j = 0; j < colsB; j++) { // bColumn
		for (int k = 0; k < colsA; k++) { // aColumn
			c[i][j] += a[i][k] * b[k][j];
		}
	}
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
}
```

2.2.2 Findings so far

According to Section 2.2.1, the piece of code can be modified using a parallel approach, in order to speed up the execution process. Also, as we mentioned above, the computation time of this process depends on 3 values; the rows of matrix A, the columns of matrix A and the columns of matrix B. For our problem, these values could be the number of patients (m), the number of medical conditions (n) and the number of factors (f).

Therefore, the matrix multiplication can be more time expensive than inverse method, due to the fact that the number of patients can grow to a huge number.

2.2.3 Next Steps

Next steps for this action include:

- Define which one(s) of the dot calculations are more expensive computationally wise;

- Convert calculation execution from sequential to parallel for the ones we identify as being more costly.

2.3. Code Kernel "Compute X and Y"

The goal of the Alternating Least Square Algorithm is to compute the matrix X and Y as shown in the algorithm below.

```
for (int iter = 0; iter<1; iter++){
      double costForPatient = 0.0;
      double costForMedicalCondition = 0.0;

      for (int i = 0; i<Parameters.NUM_OF_PATIENTS; i++){
      X[i] = handler.handlePatient(i,Y);
      costForPatient += handler.computeCostForPatient(i);
      }

      for (int j = 0; j<Parameters.NUM_OF_MEDICAL_CONDITIONS; j++){
      Y[j] = handler.handleMedicalCondition(j,X);
      costForMedicalCondition += handler.computeCostForMedicalCondition(j);
      }

      costP.add(costForPatient);
      costM.add(costForMedicalCondition);
}
```

As we can observe, we perform two different type of iterations for the computing of X and Y. The first one is by the number of patients and the second one is by the number of medical conditions. X[i] and Y[j] are row vectors with f(number of factors) values, of X and Y matrices. We compute the vector X[i] based on Y matrix and the vector Y[j] based on X matrix. And since each X[i] doesn't depend on other

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

X[k], with k!=i, and also each Y[j] depend on other Y[k], with k!=j, each step can potentially introduced to massive parallelization.

2.3.1 Specifying the kernel for heterogeneous execution

Below, we provide the piece of code that we could modify, in order to make the computation of ALS parallel.

```
for (int i = 0; i<Parameters.NUM_OF_PATIENTS; i++){
    X[i] = handler.handlePatient(i,Y,R);
    costForPatient += handler.computeCostForPatient(i);
}

for (int j = 0; j<Parameters.NUM_OF_MEDICAL_CONDITIONS; j++){
    Y[j] = handler.handleMedicalCondition(j,X,R);
    costForMedicalCondition += handler.computeCostForMedicalCondition(j);
}
```

2.3.2 Findings so far

The piece of code in Section 2.3.1 can be modified using a parallel approach, in order to speed up the execution process. Using Apache Flink, we make two kinds of partition in our dataset. The first one is partition by patient, while the second one is partition by medical condition. After that, we broadcast the matrices X and Y and make the necessary computations.

2.3.3 Next Steps

The next steps for this action is the implementation of the ALS algorithm over Apache Flink which is ongoing.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

3. NLP Use Case

The algorithmic code kernels that will be accelerated within the use case are described in detail within D.2.1 User Requirements and Data Management Plan. They include:

- Lexicographical fuzzy matching search in vocabularies using directed Acyclic Graph Words. Levenshtein distances between the dictionary words and the input words can be also used.
- Statistical fuzzy matching and classification applied in multiword expressions and/or documents using cosine similarity or TFIDF (or Okapi BM25 which is similar) applied on words or q-grams.
- Fuzzy matching of multiword expressions using Compressed Tries.

All NLP kernels have similar structure with the following characteristics:

1. There is a lexicon and an engine. The lexicon constitutes the knowledge of a domain and the engine represents ways to utilize the knowledge.
2. A common function of the engine is to take as input a word (or more, i.e. a phrase) and check the lexicon for existence. It can answer with a boolean value in case the input is in lexicon, information that the input can index (reference) or a set of fuzzy (close to input) entries of lexicon in case the input does not exist. The case of fuzzy matching, for the first three kernels we present, is also known as correction functionality and used in word processing applications in spelling and grammar checkers.
3. The usage of kernels is a two-step process. The first step loads the lexicon and engine (code) in the acceleration device. The second step is a continuous feed process where the engine gets queries (input words/phrases) and responds with information from lexicon. The functionality can be “seen” as a filter function in a stream or pipeline procedure.
4. For every class that implements the kernel code we provide a non-accelerated version of the same functionality, as well as a number of test and benchmarking programs.

The type and complexity of the lexicon and engine differentiates between kernels. We have text type lexicons as in WDK, as well as binary lexicons as in other kernels. We have simple vocabulary sets where lexicons stores only the key words as in DAWGK and map type lexicons where the key words used to index the information stored in lexicon as in CTK, TFIDFK, BM25K.

In the following sections, we present the kernels implemented with Tornado API, as well as the test programs used to validate the results and measure (benchmark) the performance.

3.1. Code Kernel “Word Distance Kernel” (WDK)

WDK has already implemented with tornado-sdk-0.1.0 version and will be used as guide for the remaining kernels.

3.1.1. Specifying the kernel for heterogeneous execution

We implemented a number of tests using different techniques for loading lexicon data and invoking the accelerated (parallel) function. The most usable approach from inside the NLP (Mnemosyne platform) is

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

presented below. First, we have the load function that loads the lexicon from a file. The file has text format and contains words one per line. The words are spread in a character buffer (array) and loaded in the device with the kernel function. In the function, we also define the result (output) buffer and warm up the device.

```
public void Load(Vector<String> dictionary)
  {
    lex_offsets = new int[dictionary.size()];
    lex_lengths = new int[dictionary.size()];
    lex_results = new int[dictionary.size()];

    for (int i = 0; i < dictionary.size(); i++)
    {
      lex_offsets[i] = lex_len;
      lex_lengths[i] = dictionary.get(i).length();

       lex_len += (lex_lengths[i] + 1);  // terminator which needed also for
cost  arrays
    }

    lex_data = new char [lex_len];
    cost     = new int  [lex_len];
    newcost  = new int  [lex_len];

    lex_len = 0;
    for (int i = 0; i < lex_offsets.length; i++)
    {
      for (int j = 0; j < dictionary.get(i).length(); j++)
        lex_data[lex_len++] = dictionary.get(i).charAt(j);
      lex_data[lex_len++] = (short) 0;
    }

    tortask.streamIn(lex_data
        , lex_offsets
        , lex_lengths
        , name_data
    ).mapAllTo(device);

    tortask.task("t0", LevenshteinDistance::LevenshteinDistance
        , lex_data
        , lex_offsets
        , lex_lengths
        , name_data
        , cost
        , newcost
        , lex_results
    );
    tortask.streamOut(lex_results);
    tortask.warmup();
  }
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Then, we present the parallel and accelerated version of Levenshtein distance function that uses the Tornado's @Parallel attribute. The function takes as input the lexicon data (lex_data), the offsets and lengths of the words stored in lexicon (lex_offsets, lex_lengths), the input word that we search in lexicon (name_data), the two (2) cost functions that are used from Levenshtein algorithm (p_cost, p_newcost) and the result array which stores the distance of the input word (name_data) and all lexicon entries. This way, we cannot only find if a lexicon contains the input word, but also identify the closest words (fuzzy matching) in case the input word is not included in the lexicon.

```
// Accelerated function with lexicon as input
  private static void LevenshteinDistance(char[] lex_data,
                                          int [] lex_offsets,
                                          int [] lex_lengths,
                                          char[] name_data,
                                          int [] p_cost,
                                          int [] p_newcost,
                                          int [] result)
  {
    int rhs_length = name_data[0];
    int rhs_offset = 1;
    char[] rhs     = name_data;
    int[] cost     = p_cost;
    int[] newcost  = p_newcost;
    int lex_count  = lex_offsets.length;
    int len1       = rhs_length + 1;

    for (@Parallel int idx = 0; idx < lex_count; idx++)
    {
      int lhs_offset = lex_offsets[idx];
      int lhs_length = lex_lengths[idx] + 1;

      // initial cost of skipping prefix in String s0
      for (int i = 0; i < lhs_length; i++)
        cost[lhs_offset + i] = i;

      // dynamically computing the array of distances

      // transformation cost for each letter in s1
      for (int j = 1; j < len1; j++)
      {
        // initial cost of skipping prefix in String s1
        newcost[0 + lhs_offset] = j;
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
        // transformation cost for each letter in s0
        for (int i = 1; i < lhs_length; i++)
        {
          // matching current letters in both strings
          int match = (lex_data[lhs_offset + (i - 1)] == rhs[rhs_offset + (j -
1)]) ? 0 : 1;

          // computing cost for each transformation
          int cost_replace = (cost[lhs_offset + (i - 1)] + match);
          int cost_insert = (cost[lhs_offset + i] + 1);
          int cost_delete = (newcost[lhs_offset + (i - 1)] + 1);

          // keep minimum cost
             newcost[lhs_offset + i] = (cost_insert < cost_delete) ?      //
Minimum(cost_insert, cost_delete, cost_replace);
              (cost_insert < cost_replace ? cost_insert : cost_replace) :
              (cost_delete < cost_replace ? cost_delete : cost_replace);
        }

        // swap cost/newcost arrays
        //int[] swap = cost; cost = newcost; newcost = swap;
        for (int k = 0; k < lhs_length; k++)
        {
          int swap = cost[lhs_offset + k];

          cost[lhs_offset + k] = newcost[lhs_offset + k];
          newcost[lhs_offset + k] = swap;
        }
      }

      // the distance is the cost for transforming all letters in both strings
      result[idx] = (cost[lhs_offset + lhs_length - 1]);
    }
  }
```

3.1.2. Findings so far

The implementation of WDK is a driver for the remaining kernels. It shares the same architecture and a similar API. We implemented a number of tests that use different techniques in order to find efficient ways to work with Tornado and acceleration. For this, we prepared examples with lexicons and test data of various sizes and calling methods. We also created a shell script that runs these tests and outputs the times of sequential and parallel (accelerated) execution of each case. The script is presented in Appendix A. It takes as option the minimum test case number. The tests are scaled from simple cases with few

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

words in lexicon and one input word to check to more complex cases with thousands of words in lexicon and tenths of search words.

3.1.3. Next Steps

Next steps include the following actions:

- Adapt to newer Tornado versions;
- Write the test programs that explore the characteristics;
- Incorporate the kernel to NLP environment (the Mnemosyne™ Natural Language platform by Neurolingo);
- Use the kernel in NLP applications.

3.2. Code Kernel "Directed Acyclic Graph Words Kernel" (DAWGK)

3.2.1. Specifying the kernel for heterogeneous execution

DAWG (Direct Acyclic Word Graph) is a graph structure where nodes represent prefix parts of words and edges continuation paths for bigger prefixes. It is a representation of Finite State Automaton (FSA) and can efficiently factorize common prefixes and suffixes of set of words. Similarly with WDK, we use a lexicon that represents the FSA and the code that traverse the automaton and responds either with a boolean success value for the existence of the input word, or a set of similar words in case that the word does not exist in the set. An important characteristic of the lexicon is that it is constructed off-line with another tool and is stored in a file as binary type.

3.2.2. Findings so far

The code of DAWGK consists of a load function which reads the automaton from the file and stores it to arrays, in order to transfer it to the device. It includes also the engine functions that take as input a word buffer, traverse the lexicon and respond either with a boolean value or a set of similar words. The load function is presented below:

```
public void LoadData(DataInputStream dat_stream, int size) throws Exception
  {
    try
    {
      aut_size       = (size / NUM_BYTES_PER_ENTRY);
      automat_edges  = new int [aut_size];
      automat_values = new char[aut_size];

      int    num_states = 0;
      byte[] buffer     = new byte[1024*NUM_BYTES_PER_ENTRY];

      while ( num_states < aut_size )
      {
         int  cbytes = (aut_size-num_states > 1024) ? 1024*NUM_BYTES_PER_ENTRY
: (aut_size-num_states)*NUM_BYTES_PER_ENTRY;
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
        dat_stream.read(buffer,0,cbytes);

        for (int i=0; i < cbytes; i += NUM_BYTES_PER_ENTRY)
        {
            automat_edges [num_states  ] = ((buffer[i] & 0x000000FF) << 24) |
((buffer[i+1] & 0x000000FF) << 16) | ((buffer[i+2] & 0x000000FF) << 8) |
(buffer[i+3] & 0x000000FF);
          automat_values[num_states++] = (char)(((buffer[i+4] & 0x000000FF) <<
8) | (buffer[i+5] & 0x000000FF));
        }
      }

      if (aut_size >= 2)
      {
        /* create a pseudo state pointing to the start state */
        start_state = automat_edges[0];
        automat_edges[0] = set_transition(automat_edges[0],automat_edges[0]);
        if (start_state < aut_size)
          return;
      }
    }
    catch (FileNotFoundException fnfex)
    {
      error(fnfex.toString());
    }
    catch (IOException ioex)
    {
      error(ioex.toString());
    }

    error("Error in input file.");
  }
```

The sequential (CPU) version of searching the automaton is presented below.

```
public boolean CheckWord(ISession session, char[] str, int start, int len)
throws Exception
  {
    int     pos;
    char    w;
    boolean found;
    int     e = 0;
    int     offset;
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
   pos = (automat_edges[0] >> POS_OF_DEST_BITS) & ~(~0 << NUM_OF_DEST_BITS);
    for (int i=start; i < start+len; i++)
    {
      if (pos > aut_size)
        error("Error in automaton file.");
      found  = false;
      w      = str[i];
      offset = 1;

      /* search the tree for current character */
      while (true)
      {
        e = automat_edges[pos + offset - 1];

        char  sym = automat_values[pos + offset - 1];

        if (sym == w)
        {
          found = true;

          break;
        }

        if (sym > w)
        {
          if ((e & (1 << POS_OF_FLAG_MASK_BIT_1)) != 0)
            break;
          offset = offset * 2;
        }
        else
        {
          if ((e & (1 << POS_OF_FLAG_MASK_BIT_2)) != 0)
            break;
          offset = offset * 2 + 1;
        }
      }

      if (!found)
        return false;
      if (i+1 < start+len)
/* if not last character in string */
        pos = (e >> POS_OF_DEST_BITS) & ~(~0 << NUM_OF_DEST_BITS);
/* get index of new state          */
    }

    return (e & (1 << POS_OF_TERM_FLAG_BIT)) != 0;
  }public boolean CheckWord(ISession session, char[] str, int start, int len)
throws Exception
  {
    int     pos;
    char    w;
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
    boolean found;
    int     e = 0;
    int     offset;

   pos = (automat_edges[0] >> POS_OF_DEST_BITS) & ~(~0 << NUM_OF_DEST_BITS);
    for (int i=start; i < start+len; i++)
    {
      if (pos > aut_size)
        error("Error in automaton file.");
      found  = false;
      w      = str[i];
      offset = 1;

      /* search the tree for current character */
      while (true)
      {
        e = automat_edges[pos + offset - 1];

        char  sym = automat_values[pos + offset - 1];

        if (sym == w)
        {
          found = true;

          break;
        }

        if (sym > w)
        {
          if ((e & (1 << POS_OF_FLAG_MASK_BIT_1)) != 0)
            break;
          offset = offset * 2;
        }
        else
        {
          if ((e & (1 << POS_OF_FLAG_MASK_BIT_2)) != 0)
            break;
          offset = offset * 2 + 1;
        }
      }

      if (!found)
        return false;
      if (i+1 < start+len)
/* if not last character in string */
        pos = (e >> POS_OF_DEST_BITS) & ~(~0 << NUM_OF_DEST_BITS);
/* get index of new state          */
    }

    return (e & (1 << POS_OF_TERM_FLAG_BIT)) != 0;
  }
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

3.2.3. Next Steps

Planning of next steps include the following actions:

- Write the acceleration function;
- Write the test programs that explore the characteristics;
- Incorporate the kernel to NLP environment;
- Use the kernel in NLP applications.

3.3. Code Kernel "Compressed Tries Kernel" (CTK)

CTK has not yet been accelerated.

3.3.1. Specifying the kernel for heterogeneous execution

The current implementation of the CTK kernel code is presented below.

```
protected void LoadData(DataInputStream stream) throws Exception
  {
    int    data_siz;

    start_state = stream.readInt();
    data_siz    = stream.readInt();
    aut_size    = stream.readInt();
    automat     = new int  [aut_size];
    autosyms    = new char [aut_size];
    autoflags   = new byte [aut_size];

    int    num_states = 0;
    byte[] buffer     = new byte[1024*7];

    while ( num_states < aut_size )
    {
              int    cbytes  =  ((aut_size-num_states  >  1024)  ?  1024*7  :
(aut_size-num_states)*7);

      stream.read(buffer,0,cbytes);

      for (int i=0; i < cbytes; i += 7)
      {
            automat   [num_states   ] = ((buffer[i] & 0x000000FF) << 24)   |
((buffer[i+1] & 0x000000FF) << 16) | ((buffer[i+2] & 0x000000FF) << 8) |
(buffer[i+3] & 0x000000FF);
        autosyms [num_states  ] = (char) ( ((buffer[i+4] & 0x000000FF) << 8) |
(buffer[i+5] & 0x000000FF) );
        autoflags[num_states++] = buffer[i+6];
      }
    }
    data = new byte[data_siz];
    stream.read(data);
  }
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Next steps for this code kernel include:

- Implementation of the accelerated version;
- Implementation of a set of test programs that will measure the efficiency of the implementation;
- Incorporation of the kernel in NLP environment (the Mnemosyne platform);
- Usage of kernel in NLP syntheses (application code).

3.4. Code Kernel "TFIDFK"

TFIDF Kernel, as well as the BM25 presented in the next section, use the same lexicon type which is a compressed tree presented in previous section. They offer different functionality than the simple CTK and sequences of words as keys rather than single words, stems or part of words called qgrams. In case that keys are sequence of words, these sequences can be viewed as documents. The information that they store is frequencies of the indexed keys in a corpus (document collection). The queries they answer take as input a similar sequence of key items and return the most relevant documents with the input sequence using the TFIDF metric. The complexity of the procedure is the volume of computation in case that the corpus we index is big and the method has to traverse big parts (or full) of the index in order to find the best matched documents. This procedure can be accelerated in devices and can be incorporated in a pipeline with other types of tasks.

Next steps include:

- Implementation of computation of TFIDF metric of a document in relation with specific corpus;
- Implementation of a set of test programs that will measure the efficiency of the implementation;
- Incorporation of the kernel in NLP environment (the Mnemosyne platform);
- Usage of kernel in NLP syntheses (application code).

3.5. Code Kernel "Best Match Kernel" (BM25K)

BM25 is a variation of TFIDF and presents better quality characteristics in the domain of data cleansing. Progress is still at very early stages. Next steps include:

- Implementation of computation of metric of a document in relation with specific corpus;
- Implementation of a set of test programs that will measure the efficiency of the implementation;
- Incorporation of the kernel in the NLP environment (the Mnemosyne platform);
- Usage of kernel in NLP syntheses (application code).

4. Green Buildings Use Case

The Spark Works Internet of Things (IoT) platform is designed to enable an easy and fast implementation of applications that utilize an IoT infrastructure. It offers high scalability in terms of users, number of connected devices and volume of data processed. The platform accommodates real-time processing of

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

information collected from mobile sensors and smartphones and offers fast analytics. The platform offers real-time processing and analysis of unlimited IoT data streams with minimal delay and processing costs. In its current Green Buildings deployment, over 400MB of data are produced daily, resulting in a yearly data volume of approximately 140GB. However, the current setup uses averaging to minimize the required storage space. For obtaining near real-time information on the building status, it is necessary to shorten the averaging period (now set to 5 minutes), an approach that will lead to significant data volume increase. In this context, the deployed IoT infrastructure will generate, handle, transfer and store a tremendous amount of data, which cannot be processed in an efficient manner using current platforms and techniques. Spark Works utilizes the E2Data stack for improving its IoT Platform in order to rapidly process the constantly accumulated data and tackle the issues generated by thousands of parallel deployments of sensors, each generating enormous data chunks that need to be processed almost in real-time. The Spark Works platform kernels that will be accelerated by the E2Data platform are the followings:

1. *Compute sum*;
2. *Compute max*;
3. *Compute min*;
4. *Compute average*.

4.1. Code Kernel "compute sum"

The kernel "compute sum" computes the summary of a batch of sensor measurements retrieved from the platform's message broker. The kernel accepts an array of primitive double values and returns the summary of the values contained in the array.

4.1.1. Specifying the kernel for heterogeneous execution

This method accepts an array of primitive double and computes the maximum double value contained in the array. The code iterates over the values of the array summarizing those values utilizing a local double variable (sum). The kernel returns the variable sum which contains the summary of the values contained in the array. This kernel is written in Java with the following implementation:

```
double computeSum(final double[] values) {
        if (Objects.isNull(values) || values.length == 0) {
            throw new IllegalStateException();
        }
        double sum = 0;
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        return sum;
}
```

4.1.2. Findings so far

The code of the kernel has been modified to meet the requirements of the current status of the E2Data execution engine. Based on this, the kernel has been adapted to a batch processing version while normally the Spark Works analytics engine is based on stream processing. Based on this, the current

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

version of the kernel accepts an array of primitive double values while according to the requirements of the analytics engine it should be ready to process a stream of composite objects.

4.1.3. Next Steps

According to the findings described in Section 4.1.2, the code of the kernel should be adapted to a stream processing version.

4.2. Code Kernel "compute max"

The kernel "compute max" computes the maximum value from a batch of sensor measurements retrieved from the platform's message broker. The kernel accepts an array of primitive double values and returns the maximum value of the array.

4.2.1. Specifying the kernel for heterogeneous execution

This method accepts an array of primitive double and computes the maximum double value contained in the array. The code initially assigns the first item of the array to a local variable (max) and then iterates over the remaining values of the array comparing each value of the loop index with the variable max. If the value of the current index is greater than the value of the variable max then it replaces the value of max with the current index value. Finally, the kernel returns the variable max which is the maximum value of the array. This kernel is written in Java with the following implementation:

```
double computeMax(final double[] values) {
        if (Objects.isNull(values) || values.length == 0) {
            throw new IllegalStateException();
        }
        double max = values[0];
        for (int i = 1; i < values.length; i++) {
            if (values[i] > max) {
                max = values[i];
            }
        }
        return max;
}
```

4.2.2. Findings so far

Our description in Section 4.1.2 for the "compute sum" kernel is valid for the "compute max" kernel too. The code has been adapted to a batch processing version utilizing an array of primitive double values.

4.2.3. Next Steps

According to the findings described in Section 4.2.2, the code of the kernel should be adapted to a stream processing version.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

4.3. Code Kernel "compute min"

The kernel "compute min" computes the minimum value from a batch of sensor measurements retrieved from the platform's message broker. The kernel accepts an array of primitive double values and returns the minimum value of the array.

4.3.1. Specifying the kernel for heterogeneous execution

This method accepts an array of primitive double and computes the minimum double value contained in the array. The code initially assigns the first item of the array to a local variable (min) and then iterates over the remaining values of the array comparing each value of the loop index with the variable min. If the value of the current index is less than the value of variable min, then it replaces the value of min with the current index value. Finally, the kernel returns the variable min which is the minimum value of the array. This kernel is written in Java with the following implementation:

```
double computeMin(final double[] values) {
        if (Objects.isNull(values) || values.length == 0) {
            throw new IllegalStateException();
        }
        double min = values[0];
        for (int i = 1; i < values.length; i++) {
            if (values[i] < min) {
                min = values[i];
            }
        }
        return min;
}
```

4.3.2. Findings so far

Once again, what described in Sections 4.1.2 and 4.2.2 is also valid for the "compute min" kernel. The code has been adapted to a batch processing version utilizing an array of primitive double values.

4.3.3. Next Steps

According to the findings described in Section 4.3.2, the code of the kernel should be adapted to a stream processing version.

4.4. Code Kernel "compute average"

The kernel "compute average" computes the average value of a batch of sensor measurements retrieved from the platform's message broker. The kernel accepts an array of primitive double values and returns the average of the array's values. The code of this kernel depends on the "compute sum" kernel since it utilizes the same code to compute the summary of the values before computing the average value.

4.4.1. Specifying the kernel for heterogeneous execution

This method accepts an array of primitive double, computes the summary of those values utilizing the "Compute sum" kernel and, finally, divides the summary with the number of values contained in the

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

provided array to produce the final result, i.e. the average value of the values contained in the provided array. This kernel is written in Java with the following implementation:

```
double computeAvg(final double[] values) {
        return computeSum(values) / values.length;
}
```

4.4.2. Findings so far

Since the code of the "compute average" depends on the code of the "compute sum" kernel and as described in Section 4.1.2, the current version of the kernel is a batch processing version with an array of primitive double values.

4.4.3. Next Steps

According to the findings described in Section 4.4.2, the code of the kernel should be adapted to a stream processing version following the next steps of "compute sum" kernel.

5. Security Use Case

iProov has developed a Java implementation of the ColourMorph kernel used in the biometric authentication pipeline. In conjunction with this, iProov has created scripts which handle file I/O and argument passing which are executed independently of the kernel. For benchmarking, random RGB input images are generated in varying sizes.

iProov has delivered the kernel source code to the Tornado team for acceleration.

5.1. Code Kernel ColourMorph

The kernel takes as input a series (n) of equal sized source RGB images containing images of faces together with a set of landmarks for each image.The kernel has a reference image again complete with a set of landmarks.The value of each pixel in the reference image is set to that of the closest pixel in each of the source images.

5.1.1. Specifying the kernel for heterogeneous execution

The Java implementation involves the use of 3-dimensional arrays of primitives. At the time of submission, Tornado did not support 3D arrays. However, the UNIMAN group expressed interest in modifying Tornado to support 3D arrays, and has been investigating this.

```
// Main method signature, arguments are 2 and 3 dimensional arrays
public static float[][][] morph(
 float[][][] inputFrame,
        float[][] inputLandmarks,
float[][][] model,
boolean[][] mask
)
```

5.1.2. Findings so far

We obtained preliminary benchmarks for the kernel, one for the unaccelerated serial Java implementation, and one for our handcrafted CUDA version. The CUDA time was lower than the Java

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

time as expected, but low-level caching is likely to be introducing noise and rendering the times imprecise. This, even despite the fact that new input images are being randomly generated for each run.

5.1.3. Next Steps

Next steps for this kernel include:

- Continue working with the Tornado team to achieve successful integration of the kernel;
- Obtain further benchmarks and make objective comparisons. Other benchmarks may include a hand-crafted OpenCL implementation, as well as Tornado accelerations with various parameters and execution plans;
- Work with the Apache Flink team to support Flink integration of the kernel.

5.2. Code Kernel PearsonR

There is not any progress recorded for the code kernel. Subsequent task activities will implement the accelerated version of the PearsonR.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Conclusions & Next Steps

All use cases have commenced their tasks towards accelerating critical code parts based on E2Data. Progress has been mainly done in the direction of developing the accelerated version, wherever that has been possible, based on the current version of Tornado. Taking preliminary measurements, was also possible on specific cases. The first preliminary development results and measurements of accelerated code kernels provide hints for the E2Data team to investigate further specific functionalities, challenging adjustment of Tornado to achieve improved performance of use case kernels. First implementation results are being described in D6.2 “Prototype V1 and preliminary evaluation”. Continuous work with iterations and interactions between use case adapters, testers and core E2Data development team, will drive the remaining work. All use cases need to improve their performance through E2Data, while progressing towards completing the overall envisaged functionality of the modified kernels in the respective use cases.

Ongoing action plans of the use cases include benchmarking on multiple load scenarios, integration with Flink, acceleration of remaining kernels and revised implementations to adapt to newer Tornado versions.

For the eHealth use case the Alternating Least Square (ALS) algorithm has been implemented in Java. Also, benchmarks for the batch implementation have been performed in order to find the points that need to accelerate. The piece of codes for the eHealth use case will be modified using a parallel approach, in order to speed up the execution process, while at the end, Flink they will be run over the modified Flink version of E2Data.

For the language processing functions of the Financial use case, the first code kernel (WDK) has been already adapted to Tornado platform. Its implementation will be a driver for the remaining kernels. A shell script that runs tests and outputs the times of sequential and parallel (accelerated) execution of each case has been already developed. Adaptation of remaining kernels, deployment of tests and integration within the language processing platform are among the next milestones of this use case. The

Green building use case will be enhanced with the stream version of the E2Data platform. Uptonow, performance improvements have been measured by adapting the use case kernels to the current version of E2Data.

The Security use case will continue working with the Tornado team to achieve successful integration of the kernel and to finally obtain further benchmarks and make objective comparisons. Other benchmarks may include a hand-crafted OpenCL implementation, as well as Tornado accelerations with various parameters and execution plans; finally, work will support Flink integration of the kernel.

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Abbreviations

- ALS: Alternating Least Squares
- API: Application Programming Interface
- BM25K: Best Match Kernel
- CTK: Compressed Tries Kernel
- DAGW: Directed Acyclic Graph Words
- HW, H/W: Hardware
- IoT: Internet of Things

Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

Appendix A

The shell script that runs tests and outputs the times of sequential and parallel (accelerated) execution of each case for WDK. It takes as option the minimum test case number. The tests are scaled from simple cases with few words in lexicon and one input word to check to more complex cases with thousands of words in lexicon and tenths of search words.

```
#!/bin/bash

export E2DATA_HOME=/projects/e2data
export JDK_HOME=${E2DATA_HOME}/jdk8
export JAVA_HOME=${JDK_HOME}

export TORNADO_SDK=${E2DATA_HOME}/tornado-sdk
export PYTHON=${E2DATA_HOME}/jython2.7.0/bin/jython
#export JAVA=${JDK_HOME}/bin/java
export JAVA=${TORNADO_SDK}/bin/tornado

export NLP_HOME=${E2DATA_HOME}/github

export
CP="${NLP_HOME}/nlp_kernels/target/nlp.tornado.test-1.0.0-SNAPSHOT.jar":${CLAS
SPATH}


#set cmd=""
#set jcmd=""
#set pcmd=""

#export TORNADO_ARGS="--printKernel --debug"
#export TORNADO_ARGS="--debug"
#export TORNADO_ARGS=-ea

export TORNADO_ARGS=

export jcmd1="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado ALEXIOS"
export jcmd2="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado2 data/Names.txt ALEXIOS"
export jcmd3="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado3 data/Names.txt data/test_words.txt"
export jcmd4="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado4 data/Names.txt data/test_words.txt"
export jcmd5="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado4 -co data/Names.txt data/test_words2.txt"

# With bigger dictionary
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
export jcmd6="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado4 data/354984si.ngl data/test_words3.txt"
export jcmd7="${JAVA} ${TORNADO_ARGS} -Duser.dir=${NLP_HOME} -Xmx2g -classpath
${CP} TestTornado4 -co data/354984si.ngl data/test_words4.txt"


export i="8"

#echo $1

case $1
in
"-1") export i="1" ;;
"-2") export i="2" ;;
"-3") export i="3" ;;
"-4") export i="4" ;;
"-5") export i="5" ;;
"-6") export i="6" ;;
"-7") export i="7" ;;
esac

if [[ "$i" -ge "1" ]]; then
  echo ${jcmd1}
  ${jcmd1}
fi

if [[ $i -ge "2" ]]; then
  echo ""
  echo ${jcmd2}
  ${jcmd2}
fi

if [[ $i -ge "3" ]]; then
  echo ""
  echo ${jcmd3}
  ${jcmd3}
fi

if [[ $i -ge "4" ]]; then
  echo ""
  echo ${jcmd4}
  ${jcmd4}
fi

if [[ $i -ge "5" ]]; then
  echo ""
  echo ${jcmd5}
  ${jcmd5}
fi

if [[ $i -ge "6" ]]; then
  echo ""
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.

```
  echo ${jcmd6}
  ${jcmd6}
fi

if [[ $i -ge "7" ]]; then
  echo ""
  echo ${jcmd7}
  ${jcmd7}
fi
```


Co-funded by the Horizon 2020 Framework Programme of the European Union under Grant Agreement nº 780245.