Universiteit Leiden
Opleiding Informatica

Using the Forelem Framework to Express
and Optimize K-means Clustering

Name:	Anne Hommelberg
Date:	21/08/2017
1st supervisor:	Harry Wijshoff
2nd supervisor:	Kristian Rietveld

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The Forelem framework was first introduced as a means to optimize database queries using optimization techniques used by compilers. Since its introduction, Forelem has proven to be more versatile and to be applicable beyond database applications. In this paper we show that the original Forelem framework can be used to express and optimize k-means clustering, thereby yielding four automatically generated implementations. These four implementations improve standard MPI C/C++ implementations of k-means as well as outperform state-of-the-art Hadoop implementations.

1 Introduction

When the Forelem framework was first introduced, it addressed the optimization of (embedded) database queries together with the wrapping C/C++ code and its associated API layer. In order to do this, database queries were automatically transformed into explicit loop structures (forelem loops), which together with the wrapping C/C++ code allowed for integral optimization. This integral optimization led to significant performance improvements of database applications [11].
The basic loop structure in the Forelem framework is the forelem loop, which iterates over a collection of tuples. Each iteration is seen as a tuple based, atomic operation, allowing the specification to be inherently parallel. In essence this yields a program specification which is free from common artifacts, like explicit data and loop structures, and associated data dependencies. For database applications this restriction proved to be very natural, but suprisingly this restriction has also proven to be natural for other applications. This is mainly caused by a side effect of an inherent property of the Forelem framework to generate data structures automatically at the end of the compile chain, so that specific characteristics of the applications are taken into account. Previous work has already shown that the Forelem framework was successful in optimizing (sparse) matrix computations [10], LU factorization [12] and PageRank [14].
In this paper, we apply the Forelem framework to k-means clustering. K-means clustering was first introduced in the fifties [13]. There are many known clustering algorithms [5], but k-means is still widely used. The algorithm provides an intuitive way to detect clusters in a set of data points. It should be noted that the number of clusters, k, is specified beforehand.
As a baseline for the experimental evaluation we used a Hadoop implementation which was developed for the scalable machine learning and datamining project: Apache Mahout [1]. This implementation is also present in the BigDataBench benchmark [6]. This benchmark contains implementations for a wide variety of problems in big data. Next to the Apache Mahout implementation we also use as a second baseline a C/C++ MPI implementation originally developed by W.-K. Liao from Northwestern University [8] and adapted for the BigDataBench benchmark by the same author. Using the Forelem framework we show that four implementations of k-means clustering can be automatically generated, which improve the latter C/C++ implementation and outperform the Apache Mahout implementation.
The Forelem framework is described in Section 3 and the k-means clustering algorithm is explained in Section 4. The k-means specification within the Forelem framework will be given in Section 4 and a proof that this specification will converge and therefore the resulting implementations will terminate is given in Section 4.2. The four implementations derived from the Forelem specification are described in Section 5 and their performance is evaluated in Section 6.

2 Related work

As the Forelem framework is based on automic, tuple based operations, it may appear similar to Linda, the tuple space coordination model [4]. Linda was introduced by David Gelernter in the eighties, using tuples as a basic operation for coordination and communication of parallel processes. The basic enabler for these operations was the fact that all tuples were stored in a physical, shared, virtual, associative memory. All these operations retrieved tuples from this memory, operated on these tuples, and stored the result tuple back in this memory. Unlike this approach, the Forelem framework does make any assumption on where these tuples are stored, instead tuples are a conceptual notion and act as a placeholder for the software optimization process of Forelem. It is this difference which enables Forelem to automatically generate data structures and their mapping into physical memory later in the optimization process. Whereby Linda suffered from performance limitations thereby hindering the widespread use for high performance parallel applications, the Forelem framework is far more versatile allowing highly efficient implementations to be derived for multiple applications.
The Forelem framework also resembles Dataflow computing. Dataflow has been a major topic in computer architecture research in the seventies and early eighties [3]. Dataflow computing is token based and at runtime these tokens are matched and computed on. Several architectures for Dataflow computing were proposed, both for the storage of tokens as well as the matching unit. For the token storage mostly a content addressable memory was foreseen. As with Linda, Dataflow computing suffered from performance issues for general use and its application, although influential, was limited to specific areas in computer hardware and software design. As such the Forelem framework can be seen as a generalization of Dataflow computing, enabling a full optmization chain for general applications.

3 Forelem

Within Forelem operations are tuple based and atomic, and are organized by the use of two different loop structures: the forelem and whilelem loops. Both structures iterate over the tuples in a tuple reservoir. These tuple reservoirs are neither physical nor virtual, but are defined on a conceptual level without specifying any order in which tuples are stored. Tuples contain either data fields or index fields.

$$\langle \texttt{DATA, I, J} \rangle$$

Index fields can be used for reference to other tuples (think of indices in database tables) or can be used to address data stored in shared spaces. As with the tuple reservoir, these shared spaces consist only at a conceptual level. With each shared space `A` an address function $\texttt{F}_{\texttt{A}}$ is associated which maps tuple indices to a unique "location" in this shared space, in which data can be stored. For the purpose of readibility, in this paper we use simple array notation `[...]` to denote these address functions. Note that actual data structures are automatically generated and optimized by the Forelem transformation engine without involvement of the programmer, therefore this notation of array indexing should not be confused with actual array data structures.
The forelem loop from which the framework derives its name, traverses all tuples in a given tuple reservoir, performing the calculation specified in the loop body. It executes the loop body exactly once for each tuple. For example, the following forelem loop computes the product of two matrices located in shared spaces `A` and `B`:

```
forelem (⟨i,j,k⟩ ∈ X)
   C[i,j] += A[i,k] * B[k,j]
```

Here the tuple reservoir X would contain tuples for every i, j, and k for which $A_{i,k} \neq 0$ and $B_{k,j} \neq 0$. Note that although this closely resembles a C-like implementation of matrix A*B multiplication, this specification is different although array notation is used to indicate the accesses to shared spaces A, B and C, see above.
The whilelem loop is an extension of the forelem loop, which continues to execute the loop body for different tuples until all of the tuples in the reservoir result in no-op operations. The loop body may consist of one or more if-statements, guarding the execution of the tuple operations. An example of a whilelem loop is shown below, which sorts the elements in a given shared space X:

```
whilelem (⟨i,j⟩ ∈ Y) {
   if (X[i] > X[j])
      swap(X[i],X[j])
}
```

Here tuple reservoir Y could contain all tuples ⟨i,j⟩ for which $0 \leq i < j < N$ with N the total number of elements to be sorted. Note that a smaller reservoir which contains only ⟨i,j⟩ such that $i = j - 1$ would also suffice, in which case the resulting implementation would closely resemble bubblesort. In fact, by choosing a specific reservoir and order in which tuples are scheduled, many existing sorting algorithms can easily be derived from this specification. Again note that X is not an array.
Note that neither of the loops specifies a specific order in which the tuples are traversed. All tuple operations are assumed to be atomic, i.e., without interference by the execution of other tuples. Also both loop structures are inherently parallel, so tuples can be visited in any order and in parallel. For the whilelem loop structure a tuple can even be executed multiple times in a row. The actual scheduling of tuple selection relies on Just Scheduling [7] (not to be confused with Just-in-Time Scheduling) on which we will not elaborate further in this paper because of page limitations. The reader is referred to a forthcoming paper which describes the formal aspects of this type of scheduling.
Once a specification is given, several transformations are applied to derive different implementations [9]. During the code generation process, the data structures used for the shared spaces will be derived automatically. Examples of these transformations will be given in Section 5, when deriving the final four implementations of the k-means clustering algorithm.

4 K-means Clustering

The k-means clustering algorithm divides a given set of data points of dimension d into k clusters. The number k is specified beforehand by the user. To start, the algorithm first initializes the k cluster centers. This can be done in various ways. A standard distribution consists of randomly assigning data points to one of the k clusters, then calculating the mean of the assigned data points to obtain the cluster center.
The algorithm consists of several iterations. During each iteration the algorithm loops over each data point, calculating the Euclidean distance to each cluster center. After each iteration, the data points are assigned to the cluster whose center was closest. After reassigning all data points, the cluster centers are set to the mean of all data points that were assigned to this cluster during this iteration. Then the next iteration starts. This can be written in pseudocode:

```
change = true
while (change) {
  change = false //assume there is no change

  //reassign data points to clusters
  for (x = 1 to N) {
    mindist = LARGE
    for (m = 1 to k)
      if (dist(DATA[x],VAL[m]) <= mindist) {
        a = m
        mindist = dist(DATA[x],VAL[m])
      }
    if (a != M[x]) {
      M[x] = a
      change = true
    }
  }

  //recalculate cluster centers
  if (change) {
    for (m = 1 to k) {
      mean, count = 0
      for (x = 1 to N)
        if (M[x] == m) {
          mean = mean + DATA[x]
          count = count + 1
        }
      VAL[m] = mean/count
    }
  }
}
```

Here `x` is a data point with `N` the total number of data points, `m` is a cluster with `k` the total number of clusters, `dist` a function that calculates the Euclidean distance, `M[x]` the cluster data point `x` is currently assigned to, `DATA[x]` the coordinates of data point `x` and `VAL[m]` the coordinates of the cluster center of cluster `m`. Note that `DATA[x]` and `VAL` are n-dimensional, and any operations involving them, including the distance function, are in fact n-dimensional operations.
Note that this algorithm will converge because the sum of all distances between data points and the center of their assigned cluster will decrease with each iteration. However, it can converge to a local minimum instead of the global minimum. A proof that the algorithm as specified within the Forelem framework converges, and therefore the whilelem loop used will terminate, is given in Section 4.2.

4.1 Forelem Specification of K-means

Recall that the Forelem framework allows "random" execution of the (tuple) operations in any order and for an arbitrary amount of times, in contrast to this classic implementation of k-means which explicitly determines the order of operations. The key difference between the

Forelem implementations and other parallel implementations for k-means clustering will be that to arrive at the Forelem specification, the computation must be reduced to its core.
To do so we first note that the classic algorithm consists of a main while loop which continues operations until no change is made. This naturally corresponds to using a whilelem loop in the Forelem specification, which by definition terminates as soon as all tuples result in a no-op operation. In fact, the resulting Forelem specification will use only a single outer whilelem loop.
Next we note that the loop body of the classic algorithm is split into two separate steps: reassigning the data points and recalculating the cluster centers. For both steps the classic algorithm contains a 2-dimensional for loop, looping over each possible combination of a data point and a cluster. In the first step, the distance between the data point and the cluster is compared to the best recorded distance and if this distance is smaller, the best recorded distance is updated. In the second step, the data point is then taken into account when recalculating the cluster center, only if the distance to the given cluster is the best recorded distance.
For the specification of the whilelem loop these two steps are merged into one, thereby removing the necessary bookkeeping such as the change variable seen in the classic algorithm. After all, these variables do not contribute to the essence of the computation in the classic algorithm. At the same time this merger results in the removal of the artificial barrier between the two steps, resulting in a single whilelem loop in which all steps of the two separate inner for loops are combined into single point operations. In order to ensure that these point operations can be executed in a random fashion and independetely of each other, the following observations are used:

1. The first step we need for the whilelem loop body can be captured as a simple if-statement: if the distance between a data point `x` and a cluster `m` is smaller than the best recorded distance, i.e., the distance to the cluster a data point is currently assigned to (`M[x]`), then we must reassign this data point. If not, then no operation is needed, as is also captured by the use of a boolean recording whether a change occurred in the classic implementation.

2. The first observation combined with the observation that if `M[x] == m` then clearly the distance will not be strictly smaller, gives us the condition of the if-statement in the whilelem loop:

   ```
   if (M[x] != m && dist(DATA[x],VAL[m]) < dist(DATA[x],VAL[M[x]])) { ... }
   ```

   where `dist` calculates the Euclidean distance.

3. The reassigning of the data point `x` can then be captured in the body of the if-statement by simply stating `M[x] = m`.

Therefore the whilelem loop in the Forelem specification will need to loop over each combination of a cluster `m` and a data point `x`, and take the same steps in the loop body as the classic algorithm. Our reservoir `T` will therefore contain tuples $\langle$`m,x`$\rangle$. Note that the core idea behind the k-means clustering algorithm is the fact that the cluster centers are equal to the mean of all data points assigned to a cluster. In fact, this is where it derives its name. For the classic algorithm, this is the case both at the start of each iteration of the outer while loop. Assuming that this is true at the start of an execution of the whilelem loop body, we can derive a simple formula for updating the cluster centers after reassigning a single point, to

ensure that this will still be the case for the next iteration. Note that to regain the sum of all data points assigned to a cluster, the current center can simply be multiplied by the number of data points assigned to this cluster. From this sum we can then subtract the data point that is being reassigned for `M[x]`, the cluster it belonged to before, and add the data point to the sum for the new cluster `m`. The result can be divided by the new size of the clusters to obtain their new cluster centers.
In essence, k-means clustering can therefore be captured by the following Forelem pseudocode:

```
whilelem (⟨m,x⟩ ∈ T) {
  if (M[x] != m && dist(DATA[x],VAL[m]) < dist(DATA[x],VAL[M[x]])) {
    VAL[M[x]] = (VAL[M[x]]*SIZE[M[x]] - DATA[x]) / (SIZE[M[x]] - 1)
    SIZE[M[x]] -= 1
    VAL[m] = (VAL[m]*SIZE[m] + DATA[x]) / (SIZE[m] + 1)
    SIZE[m] += 1
    M[x] = m
  }
}
```

Here `x` is a data point and `m`, `M[x]` is the cluster `x` is currently assigned to, `dist` calculates the Euclidean distance, `DATA[x]` is the coordinates of data point `x` and `VAL[m]` and `SIZE[m]` are the cluster center and size of a cluster `m` respectively. Note again `DATA[x]` and `VAL[m]` are n-dimensional and all operations that involve them are in fact n-dimensional operations, including the distance function `dist`. These operations have been abbreviated to improve readibility.

Algorithm 1 The initial Forelem specification of k-means clustering.

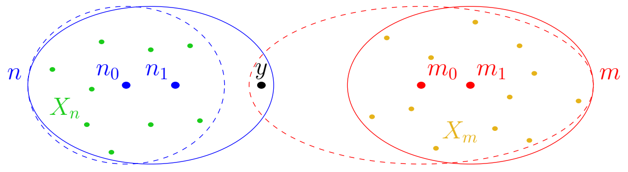
```
whilelem (⟨m,x⟩ ∈ T) {
   if (M[x] != m) {
      distold = 0
      forelem (⟨n,y,i⟩ ∈ Tₑ.⟨n,y⟩[⟨m,x⟩])
         distold += (VAL[M[y],i] - DATA[x,i])**2
      distnew = 0
      forelem (⟨n,y,i⟩ ∈ Tₑ.⟨n,y⟩[⟨m,x⟩])
         distnew += (VAL[M[y],i] - DATA[x,i])**2
      if (distnew < distold)
         forelem (⟨n,y,i⟩ ∈ Tₑ.⟨n,y⟩[⟨m,x⟩])
            VAL[M[y],i] = (VAL[M[y],i] * SIZE[M[y]] - DATA[y,i])
                                    / (SIZE[M[y]] - 1)
         SIZE[M[x]] -= 1
         forelem (⟨n,y,i⟩ ∈ Tₑ.⟨n,y⟩[⟨m,x⟩])
            VAL[n,i] = (VAL[n,i] * SIZE[n] + DATA[y,i]) / (SIZE[n] + 1)
         SIZE[m] += 1
         M[x] = m
      }
   }
}
```

Note that to capture both the reassigning of data points and the recalculation of the cluster centers within a single whilelem loop, the recalculation of a cluster center is now done

immediately after assigning a data point instead of after all data points are reassigned. All computation that is done for a specific data point and cluster is thus executed together leading to a natural parallellization. There is no need for the programmer to further optimize the implementation, since the Forelem framework will do so automatically. The compiler is now no longer restricted to a certain order of operations, and artifacts of this restriction such as the boolean recording whether a change occurred have been removed.
For the final, full Forelem specification we need to define how the shared spaces are initialized, as well as extend the tuple reservoir `T` to allow operating on the coordinates of the data points and cluster centers. The `DATA` shared space is initialized when reading the data points from file. The `M` shared space is initialized uniform random for each data point. The `SIZE` and `VAL` shared spaces are then initialized accordingly, setting the cluster centers to the mean of all assigned data points.
The extension of tuple reservoir `T` is defined as tuple reservoir $\mathtt{T_e}$ consisting of all tuples $\langle\mathtt{n,y,i}\rangle$ where $0 \leq \mathtt{n} < \mathtt{k}$ is one of the clusters, $0 \leq \mathtt{y} < \mathtt{N}$ one of the data points with `N` the total number of data points, and $0 \leq \mathtt{i} < \mathtt{d}$ an index with `d` the dimension of the data points. The main loop will traverse the reduced reservoir `T` consisting of all tuples $\langle\mathtt{m,x}\rangle$ with `m` a cluster and `x` a data point as before. We can then use the extended reservoir $\mathtt{T_e}$ to iterate over all coordinates of a vector, where $\mathtt{T_e.}\langle\mathtt{n,y}\rangle\mathtt{[}\langle\mathtt{m,x}\rangle\mathtt{]}$ is notation for selecting those tuples $\langle\mathtt{n,y,i}\rangle \in \mathtt{T_e}$ such that `n == m` and `y == x`.
Using all these elements we can specify the k-means clustering algorithm using a single whilelem loop. The basic Forelem specification of k-means clustering is given in Algorithm 1.

4.2 Correctness

As mentioned before, the convergence of the k-means clustering algorithm relies on the fact that the sum of all distances between data points and the center of their assigned cluster decreases with each iteration. Since the cluster centers are recalculated with each reassignment of a data point, we will prove that this sum will be strictly decreasing in this case as well. We pose the theorem that the algorithm given in Algorithm 2 converges.
To prove this statement, first note that since the total number of data point is finite, and therefore the total number of possible assignments to clusters is finite as well, if the sum is indeed strictly decreasing this proves that the algorithm converges. Also note that it is possible for the Forelem specification of k-means clustering to converge to a local minimum instead of the desired global minimum. As is the case with all k-means clustering implementations, no provisions are taken to prevent this.
Assume that the loop body in Algorithm 2 is being executed for a tuple $\langle n, y\rangle$ and the inner conditional statement is satisfied, i.e., the data point y is being reassigned to cluster n. We define several variable names as depicted by the following figure.



Let m be the cluster y is currently (at the start of the loop body) assigned to. Let n_0 be the current cluster center of n and m_0 the current cluster center of m. Since the cluster centers will change during the execution of the loop body, we denote their new values with n_1 and m_1

respectively. Finally, let X_m be the set of data points currently assigned to cluster m, except y. Let X_n be the set of data points currently assigned to cluster n.
We are interested in the sum of all distances between all data points and their associated cluster centers. Let X be the set of all data points and denote the cluster center associated with an $x \in X$ as c_x, then we can write the sum as:

$$\sum_{x \in X} d(c_x, x) \tag{1}$$

where d is the Euclidean distance measure.
Note that after executing the loop body, only some of the distances in the sum given in Equation 1 will change. Namely, the distance of y to its associated cluster center will change and since the cluster centers of m and n will change, the distances for all points in the sets X_m and X_n will change. Therefore, the part of the sum we are interested in can be written as:

$$d(m_0, y) + \sum_{x \in X_n} d(n_0, x) + \sum_{x \in X_m} d(m_0, x) \tag{2}$$

Since the tuple $\langle y, n \rangle$ satisfies the condition of the inner if-statement in Algorithm 2, we know that $d(m_0, y) > d(n_0, y)$. Thus showing that Equation 2 is strictly greater than:

$$d(n_0, y) + \sum_{x \in X_n} d(n_0, x) + \sum_{x \in X_m} d(m_0, x) \tag{3}$$

Note that the mean of all data points minimizes the sum of Euclidean distances from the data points to a single point. This is in fact the main reason that the Euclidean distance must be used for the k-means clustering algorithm to converge. Since n_1 is the mean of the set $X_n \cup \{y\}$ we can thus conclude that:

$$d(n_0, y) + \sum_{x \in X_n} d(n_0, x) \geq d(n_1, y) + \sum_{x \in X_n} d(n_1, x) \tag{4}$$

And similarly:

$$\sum_{x \in X_m} d(m_0, x) \geq \sum_{x \in X_m} d(m_1, x) \tag{5}$$

Thus showing that Equation 3 is greater than or equal to:

$$d(n_1, y) + \sum_{x \in X_n} d(n_1, x) + \sum_{x \in X_m} d(m_1, x) \tag{6}$$

Therefore the sum in Equation 1 strictly decreases each time a data point is reassigned, showing that Algorithm 2 converges.

5 Transformations and Implementations

The specification for the k-means clustering algorithm given in Algorithm 1 captures the essence of the algorithm and is used as a starting point to derive several implementations. In this section we will describe the transformations used to derive the final four implementations used in Section 6 to evaluate the performance of the implementations derived using the Forelem framework. It should be noted that more transformations, and many more implementations are possible.

5.1 Multi-tuple Operators

The Forelem framework includes three multi-tuple operators: map, reduce and assign. These operators are designed to allow stating simple vector operations in a single line of code, instead of using forelem loops. For example, for our k-means specification they can be used to calculate the Euclidean distance between two vectors of any dimension and to update the cluster centers, removing the four inner forelem loops shown in Algorithm 1.

These operators require an enumerator, which is a special function that will enumerate over all tuples in a reservoir which satisfy a given condition. For k-means clustering, given the tuple reservoir $\mathtt{T}$ containing tuples $\langle\mathtt{m,x}\rangle$ and the tuple extended reservoir $\mathtt{T_e}$ containing tuples $\langle\mathtt{n,y,i}\rangle$, we define enumerator $\mathtt{enum_T}(\langle\mathtt{m,x}\rangle)$ which enumerates over all $\langle\mathtt{n,y,i}\rangle \in \mathtt{T_e}$ such that `n == m` and `y == x`. Note that the order in which the tuples are given by the enumerator is not fixed, similar to the forelem loop structure the multi-tuple operators are inherently parallel.

The map operator takes a (mathematical) function and an enumerator as its arguments. The function provided should take a tuple returned by the enumerator as its input. The map operator then maps each tuple returned by the enumerator to a corresponding value through the function. For example, given the enumerator $\mathtt{enum_T}(\langle\mathtt{m,x}\rangle)$, we define a map operator that sends each tuple to their contribution to the Euclidean distance between `m` and `x`:

```
map(⟨n,y,i⟩ ∈ Tₑ -> ((VAL[n,i] - DATA[y,i])**2), enum_T(⟨m,x⟩))
```

The reduce operator takes an operation such as + and a map as its arguments, applying the given operation to each value returned by the map. Given the map operation, we can use the reduce operation to calculate the squared Euclidean distance between `m` and `x` in a single line statement:

```
reduce(+, map(⟨n,y,i⟩ ∈ Tₑ -> (VAL[n,i] - DATA[y,i])**2, enum_T(⟨m,x⟩)))
```

The assign operator takes a function specifying shared space variables, a function specifying values and an enumerator as its arguments. Again each function should take an element returned by the enumerator as its argument, mapping it to a variable in a shared space and a value respectively. The assign operator then assigns the corresponding value to the shared space variable for each element returned by the enumerator. For example, updating the cluster center of cluster `m` after reassigning data point `x` to it can now be written as:

```
assign(⟨n,y,i⟩ ∈ Tₑ -> VAL[n,i], ⟨n,y,i⟩ ∈ Tₑ ->
       (VAL[n,i] * SIZE[n] + DATA[y,i]) / (SIZE[n] + 1), enum_T(⟨m,x⟩))
```

Note that while the order in which tuples are returned by the enumerator is not fixed, therefore the order in which the multi-tuple operators operate is not fixed either, the assign operator does guarantee that the same tuple will be used on both sides of each assignment. It is a basic extension of the map operator, which only specifies a value to which a tuple is mapped, but not a location to which this value is to be assigned.

Using these multi-tuple operators, we rewrite the Forelem specification given in Algorithm 1 to the Forelem specification shown in Algorithm 2. Note that the difference between these two specifications is mainly syntactic, abbreviating the inner forelem loops to put more emphasis on the core calculation.

Note that although the naming convention suggests similarity with the MapReduce framework, the Forelem map and reduce operator are essentially different. The map and reduce operators are merely used as a means to specify vector operations in a single line statement, eliminating the need for a forelem loop. They are not used to specify separate tasks which are then to be executed by separate processes.

Algorithm 2 The Forelem specification of k-means clustering using multi-tuple operators.

```
whilelem (⟨m,x⟩ ∈ T) {
   if (M[x] != m) {
      distold = reduce(+, map(⟨n,y,i⟩ ∈ $T_e$ -> (VAL[M[y],i] - DATA[y,i])**2,
                        $enum_T$(⟨m,x⟩)))
      distnew = reduce(+, map(⟨n,y,i⟩ ∈ $T_e$ -> (VAL[n,i] - DATA[y,i])**2,
                        $enum_T$(⟨m,x⟩)))
      if (distnew < distold) {
         assign(⟨n,y,i⟩ ∈ $T_e$ -> VAL[M[y],i], ⟨n,y,i⟩ ∈ $T_e$ -> (VAL[M[y],i]
                * SIZE[M[y]] - DATA[y,i]) / (SIZE[M[y]] + 1), $enum_T$(⟨m,x⟩))
         SIZE[M[x]] -= 1
         assign(⟨n,y,i⟩ ∈ $T_e$ -> VAL[n,i], ⟨n,y,i⟩ ∈ $T_e$ ->
                (VAL[n,i] * SIZE[n] + DATA[y,i]) / (SIZE[n] + 1), $enum_T$(⟨m,x⟩))
         SIZE[m] += 1
         M[x] = m
      }
   }
}
```

5.2 Loop Blocking

The Forelem framework provides an inherently parallel specification. To divide the tuples in the tuple reservoir among processors, loop blocking can be used. Since this may cause certain shared spaces to only be used by a single process, this can also be used to reduce the amount of variables that need to be shared among processes. Given a tuple reservoir T, we split it in parts, allowing each process to iterate over their part of the reservoir T_p. Ideally each process is then executed in parallel. Note that the split should satisfy $\bigcup_k T_p = T$. Depending on the program that is to be executed, different splits may be optimal. Typically, this transformation is used last and the split is chosen such that at least one of the shared spaces does not need to be shared among processors. An example will be given in the next section.

5.3 Orthogonalisation

The orthogonalization transformation can be used to optimize the order in which tuples are visited. It introduces an outer loop, which adds an order to the processing of the tuples. The outer loop selects one or more fields of the tuples, the inner loop then loops over those tuples in the original reservoir which contain the selected values for these fields. The loop blocking transformation can then be applied to this outer loop. The result is that tuples are now processed in particular groups, and the loop blocking split can be made based on the values for certain fields of the tuple.
In our case, we choose to let the outer loop iterate over all data points and the inner loop iterate over all clusters. This then allows us to apply loop blocking, which results in each process needing only the DATA and M values that apply to its own data points. This leads to the first implementation of the k-means clustering algorithm as shown in Algorithm 3.
Here we denote the tuple reservoir containing all data points as X, and the different parts assigned to each process as X_p. Also note that T.⟨x⟩[⟨y⟩] is notation to select all tuples

$\langle$m,x$\rangle \in$ T such that y == x.

Algorithm 3 The Forelem specification of k-means clustering after using orthogonalization and loop blocking. Note that the loop body is identical to the loop body of Algorithm 2.

```
whilelem (⟨y⟩ ∈ X_p) {
   forelem (⟨m,x⟩ ∈ T.⟨x⟩[⟨y⟩]) {
      if (M[x] != m) {
         distold = reduce(+, map(⟨n,y,i⟩ ∈ T_e -> (VAL[M[y],i] - DATA[y,i])**2,
                           enum_T(⟨m,x⟩)))
         distnew = reduce(+, map(⟨n,y,i⟩ ∈ T_e -> (VAL[n,i] - DATA[y,i])**2,
                           enum_T(⟨m,x⟩)))
         if (distnew < distold) {
            assign(⟨n,y,i⟩ ∈ T_e -> VAL[M[y],i], ⟨n,y,i⟩ ∈ T_e -> (VAL[M[y],i]
                   * SIZE[M[y]] - DATA[y,i]) / (SIZE[M[y]] + 1), enum_T(⟨m,x⟩))
            SIZE[M[x]] -= 1
            assign(⟨n,y,i⟩ ∈ T_e -> VAL[n,i], ⟨n,y,i⟩ ∈ T_e -> (VAL[n,i]
                   * SIZE[n] + DATA[y,i]) / (SIZE[n] + 1), enum_T(⟨m,x⟩))
            SIZE[m] += 1
            M[x] = m
         }
      }
   }
}
```

5.4 Localization

To take advantage of the cache used in modern computers, it can be beneficial to increase data locality. Instead of storing the data points and their associated clusters separately in shared spaces, the localization transformation allows us to include these as fields in the tuple.
Using the localization transformation (followed by orthogonalization and loop blocking) we obtain a different specification shown in Algorithm 4. A tuple $\langle x_0, x_1, ..., x_{d-1}, c_x \rangle$ now contains the value of a data point $(x_0, x_1, ..., x_{d-1})$ with d the dimension, and the associated cluster as c_x.
The application of the localization transformation causes different data structures to be generated during the derivation of the implementations. For instance, for the implementations derived from the specification shown in Algorithm 3 the different shared spaces will be stored in separate arrays. For implementations derived from this new specification in Algorithm 4, the information previously stored in the shared spaces DATA and M will be stored in an array of structs instead. The loop blocking transformation still allows us to distribute this array over processors.
Note that the tuples t' in the extended reservoir T_e now also consist of more fields: t' $=$ $\langle$n,y_0,y_1, ... ,y_{d-1},c_y,i$\rangle$. They are abbreviated to t' in Algorithm 4 to improve legibility.

Algorithm 4 The Forelem specification of k-means clustering after using orthogonalization, localization and loop blocking. Note that all `t'` $\in$ $\mathtt{T_e}$ are abbreviated and `t'` $= \langle$n,y_0,y_1, ... ,y_{d-1},c_y,i$\rangle$.

```
whilelem (⟨z_0,z_1, ... ,z_{d-1},c_z⟩ ∈ X_p) {
   forelem (t = ⟨m,x_0,x_1, ... ,x_{d-1},c_x⟩ ∈
                     T.⟨x_0,x_1, ... ,x_{d-1},c_x⟩[⟨z_0,z_1, ... ,z_{d-1},c_z⟩]) {
      if (c_x != m) {
         distold = reduce(+, map(t' ∈ T_e -> (VAL[c_y,i] - y_i)**2, enum_T(t)))
         distnew = reduce(+, map(t' ∈ T_e -> (VAL[n,i] - y_i)**2, enum_T(t)))
         if (distnew < distold) {
            assign(t' ∈ T_e -> VAL[c_y,i], t' ∈ T_e -> (VAL[c_y,i] * SIZE[c_y] - y_i)
                   / (SIZE[c_y] - 1), enum_T(t))
            SIZE[c_x] -= 1
            assign(t' ∈ T_e -> VAL[n,i], t' ∈ T_e -> (VAL[n,i] * SIZE[n] - y_i)
                   / (SIZE[n] - 1), enum_T(t))
            SIZE[m] += 1
            c_x = m
         }
      }
   }
}
```

5.5 Communicating the Cluster centers

We have now given the two k-means clustering specifications used for our final implementations in Algorithm 3 and Algorithm 4 respectively. In these specifications communication of partial results between processors is explicitly not specified. In fact, this code is generated and inserted automatically during the code generation process, in which a Forelem specification is translated to an executable code. Any order of execution for the tuples that has been established during the derivation of the Forelem specification is preserved and taken into account for further optimization steps.

In both implementations, the (local) cluster center and size is updated each time the loop body is traversed and a data point is reassigned. Based on this, we now describe two possible communication codes that are generated during the code generation.

The first option simply recalculates the means of the clusters. In this case, the communication is directly derived from the initialization procedure described for the `VAL` and `SIZE` shared spaces:

```
forelem (⟨x⟩ ∈ X) {
   SIZE[M[x]] += 1
   forelem (⟨n,y,i⟩ ∈ T_e.⟨n,y⟩[⟨M[x],x⟩])
      VAL[n,i] += DATA[y,i]
}
forelem (⟨m,i⟩ ∈ M)
   VAL[m,i] = VAL[m,i] / SIZE[m]
```

Here `X` and $\mathtt{T_e}$ are the same tuple reservoirs as before and `M` is a tuple reservoir containing tuples $\langle$m,i$\rangle$ with $0 \le \mathtt{m} < k$ a cluster and $0 \le \mathtt{i} < \mathtt{d}$ the index of a coordinate of its center.

Both the `VAL` and `SIZE` shared spaces are first initialized to 0, and the `M` shared space has already been initialized.
This initialization procedure is inserted within the inner loop and made to operate on a partition of the tuple rservroi $\mathtt{X}_\mathtt{p}$. By doing so, the loop is automatically parallelized, recalculating the cluster centers and sizes using only the values of `M` and `DATA` stored locally by each processor. Each processor adds the data vectors associated with each cluster to each other. Because this computation was parallelized, a reduction code is inserted which makes all processors communicate the sums and local sizes of each cluster. This ensures the results are equal to the non-parallelized version and allows each processor to calculate the (global) mean. We will refer to this as the recalculation communication scheme.
The second option utilizes the adapted cluster centers and is derived from the local calculations made by each process. Each process makes (small) adaptations to the cluster centers during the iteration. The processes communicate the new cluster centers by exchanging which data points were reassigned and each process then mimicking the recalculation made by the process that originally reassigned the data point. However, all these (small) recalculations can be optimized by combining them with each other. From this a formula to calculate the new cluster centers and sizes using only the old centers and sizes from the start of the calculation and the new local cluster centers and sizes can be derived.
Let C_i and S_i be the cluster center and size of cluster i before the iteration. Let C_{ij} and S_{ij} be the local cluster center and cluster size for process j after the iteration. Then the new global cluster centers C'_i and cluster sizes S'_i can be obtained through the following formulas:

$$S'_i = \sum_j S_{ij} - (p-1) \cdot S_{i0} \tag{7}$$

$$C'_i = \frac{\sum_j (S_{ij} \cdot C_{ij}) - (p-1) \cdot C_i \cdot S_i}{S'_i} \tag{8}$$

were p is the total number of processes. We will refer to this as the derived communication scheme.
To obtain the four implementations used in the experiments, we combine these two different communication schemes with the two different specifications given in Algorithm 3 and Algorithm 4.

6 Experiments

To evaluate the performance of the derived k-means clustering implementations we ran several experiments, using two implementations from the BigDataBench benchmark as a baseline. We will first briefly discuss these implementations in Section 6.1. The experimental setup is explained in Section 6.2 and the results are given in Section 6.3.

6.1 K-means in BigDataBench

The BigDataBench benchmark [6] contains several parallel implementations of the k-means clustering algorithm. For our first baseline we choose to use the Hadoop implementation, which uses the implementation included in the Apache Mahout project [1]. Hadoop provides a way to easily parallelize existing algorithms, which is something the Forelem framework also wishes to achieve. We will refer to this implementation as the Hadoop implementation.

Since the final implementations generated using the Forelem framework will use C/C++ code and MPI, we also use the C/C++ MPI implementation from the BigDataBench benchmark. This implementation is an adaptation of the implementation written by W.-K. Liao [8]. It takes a more traditional approach to parallellizing k-means clustering, first all processes reassign the data points in parallel, then they recalculate the cluster centers in parallel. Recall that the main difference between this approach and the Forelem implementations, will be that Forelem has interleaved the reassigning of data points and recalculation of the cluster centers. The Forelem implementations are also derived and optimized automatically from the initial specification. While the techniques used by this implementation and the Forelem implementations are similar, the path to the implementations was not. We will refer to the C/C++ MPI implementation from the BigDataBench benchmark as BigDataBench MPI.

6.2 Experimental Setup

For our experiments we have chosen to write a random data generator to exclude any bias towards initial distributions or other artifacts. The generator is given a number of data points to generate, a dimension of the desired data points and a number of clusters to generate the data in. It first generates the intended cluster centers using a uniform distribution in the interval $[0, 10]$ and generates a standard deviation for each cluster, uniform random in the interval $[\frac{10}{16}, \frac{10}{8}]$. To generate a data point, the generator first uniform randomly chooses a cluster to assign it to, then uses a normal distribution with the generated center as a mean and the generated standard deviation. Note that it is possible for coordinates of the generated data points to fall outside the interval $[0, 10]$. All data sets used in the experiment contained data points of dimension 4, generated in 4 clusters. For each implementation the data is stored in ASCII format, since the format for the BigDataBench MPI and Hadoop implementation differ slightly the Forelem implementations can read both formats.
Throughout this section, we will refer to the implementation corresponding to Algorithm 3 using the recalculation communication scheme as Implementation 1. Implementation 2 also corresponds to Algorithm 3, but uses the derived communication scheme. Similarly, Implementation 3 and Implementation 4 correspond to Algorithm 4 and use the recalculation and derived communication scheme respectively.
Note that the Hadoop implementation uses a convergence delta to determine whether the process has converged. If the change in the cluster centers during an iteration is less than this convergence delta, the calculation terminates. To allow a fair comparison, this convergence delta was added to Implementation 1 to 4. Similarly, the BigDataBench MPI implementation uses a threshold to determine convergence. If the fraction of data points that are switched to a different cluster center during an iteration is less than the given threshold, calculation terminates. This was also added to Implementation 1 to 4.
The experiments ran on (up to) 8 nodes of the DAS-4 cluster [2]. A node in this cluster consists of 2 CPU sockets, each containing a 4-core CPU with Hyper-Threading. This yields a total of 8 physical cores and 16 virtual cores per node. Implementation 1 to 4 were run in several different configurations, varying between 1 to 8 nodes and 1 to 8 threads per node. The Hadoop implementation was run using the provided set-up on the cluster which allowed it to use up to 8 nodes. The BigDataBench MPI implementation was run on a configuration of 8 nodes each containing 8 threads to provide a comparison with both the Forelem implementations and the Hadoop implementation.
Note that due to the high influence of the initial cluster centers on the time needed for the

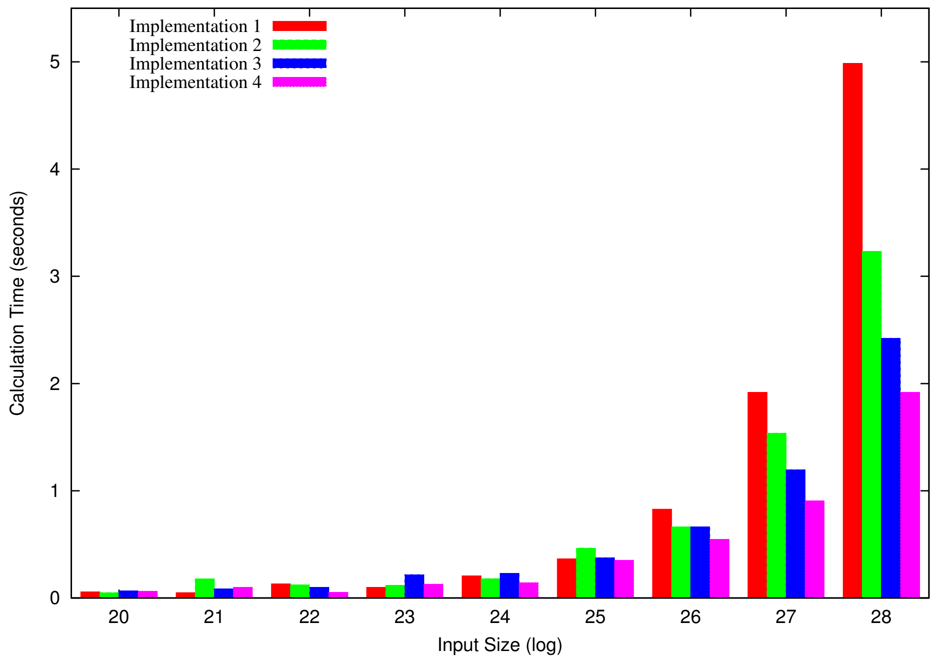


Figure 1: The calculation time of Implementation 1 to 4 on the 8×8 configuration, using a convergence delta of 0.0001.

process to converge, the results for Implementation 1 to 4 and the Hadoop implementation varied between runs. The BigDataBench MPI implementation uses a deterministic initialization, selecting the first k data points in the input file as the inital cluster centers. Its results show a similar variance when running on shifting the data points in an input file to force a different choice for the initial cluster centers. To mitigate the influence of the initialization, all results shown are averages over 10 runs.

6.3 Results

In our experiments, we first focus on the performance of the forelem framework implementations for different parameters such as input size and the number of threads. Then, we will use the two implementations taken from the BigDataBench benchmark, the BigDataBench MPI implementation and the Hadoop implementation, to provide a baseline to compare the performance of Implementation 1 to 4 to.

For the first experiment, we ran Implementation 1 to 4 on the 8×8 configuration for data sets containing 2^{20} to 2^{28} data points, the results are shown in Figure 1. These results show that applying localization, as is done for Implementation 3 and 4, decreases the calculation time. Similarly, for larger data sets, the calculation time is decreased by using the derived communication scheme instead of the recalculation communication scheme, as shown by Implementation 2 and 4 perfoming better than Implementation 1 and 3 respectively. Both effects becomes more apparent on the larger data sets. Note that the calculation time shown excludes the time needed for input and output. We focus our experiments on the part of the code that was specified and optimized in the forelem framework: time is measured from the start of the initialization of the cluster centers until the execution of the whilelem loop terminates.

In a second experiment, Implementation 1 to 4 were run on configurations containing 1 to 8 nodes, using 1, 2, 4 or 8 threads per node and the data set containing 2^{26} data points. The

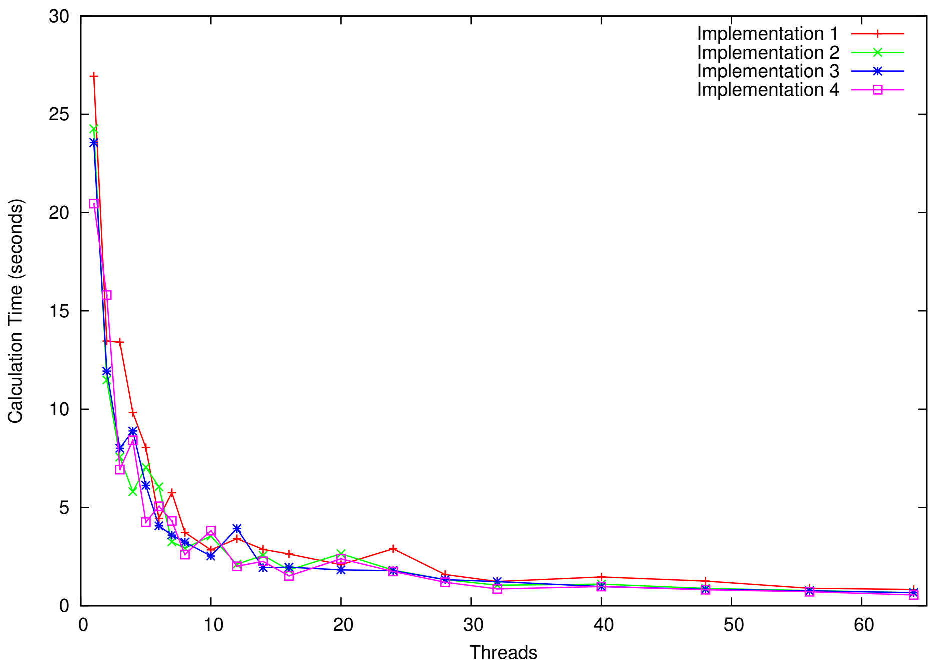


Figure 2: The calculation time of Implementation 1 to 4 for varying numbers of threads, using a convergence delta of 0.0001.

calculation time for the four implementations using different numbers of threads are shown in Figure 2. Note that when different configurations would yield the same number of threads, the configuration using the lowest number of nodes was used. For example, configurations 2×4 and 1×8 both yielded 8 threads, and configuration 1×8 was used in Figure 2. The results show that when the number of threads double, the calculation time becomes roughly half of the calculation time, thus showing that the implementations scale very well. Due to the range needed to show all results in Figure 2 the graph may appear to approach a limit for the higher number of threads, but it in fact continues to go down at a similar rate as before. From 32 to 64 threads, the calculation time decreases with a factor 1.6 on average.

Finally, to further investigate the behaviour of Implementation 1 to 4 we also ran an experiment using data sets with different dimensions and numbers of clusters. Both experiments were run on data sets of size 2^{26}. The results for Implementation 1 to 4 when running on data sets with $k = 4$ and different dimensions are shown in Figure 3. These results show that the calculation time slightly increases when the dimension does, which is due to the increase in operations needed to calculate the Euclidean distance and recalculate the cluster centers. However, the increase in calculation time is very small compared to the increase in dimension: an increase of a factor 8 in dimension only results in an increase of about a factor 2 in calculation time.

The results for Implementation 1 to 4 when running on data sets with dimension 4 and different numbers of clusters k are shown in Figure 4. Similar to the experiment with different dimensions, the calculation time appears to increase slightly as the number of clusters increases. This is due to an increased amount of information needing to be communicated, when the processes communicate the cluster centers. However, since the frequency of this communication does not increase, only the length of the messages does, the increase in calculation time is small.

From these first experiments we can conclude that the Forelem framework implementations scale well. The additional optimizations applied in Implementation 4 indeed improve the per-

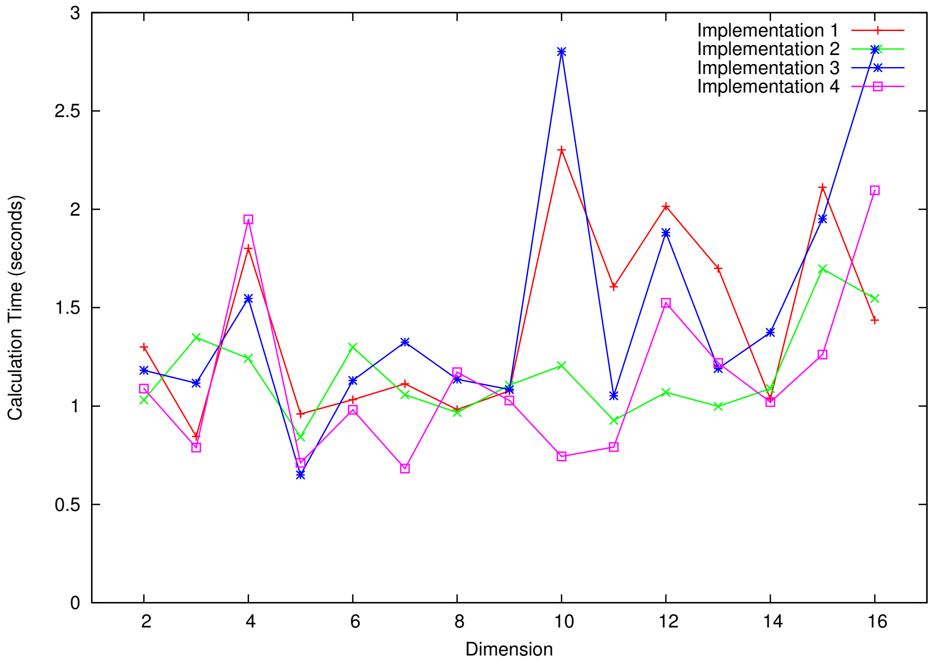


Figure 3: The calculation time of Implementation 1 to 4 for different dimensions, using a convergence delta of 0.0001.

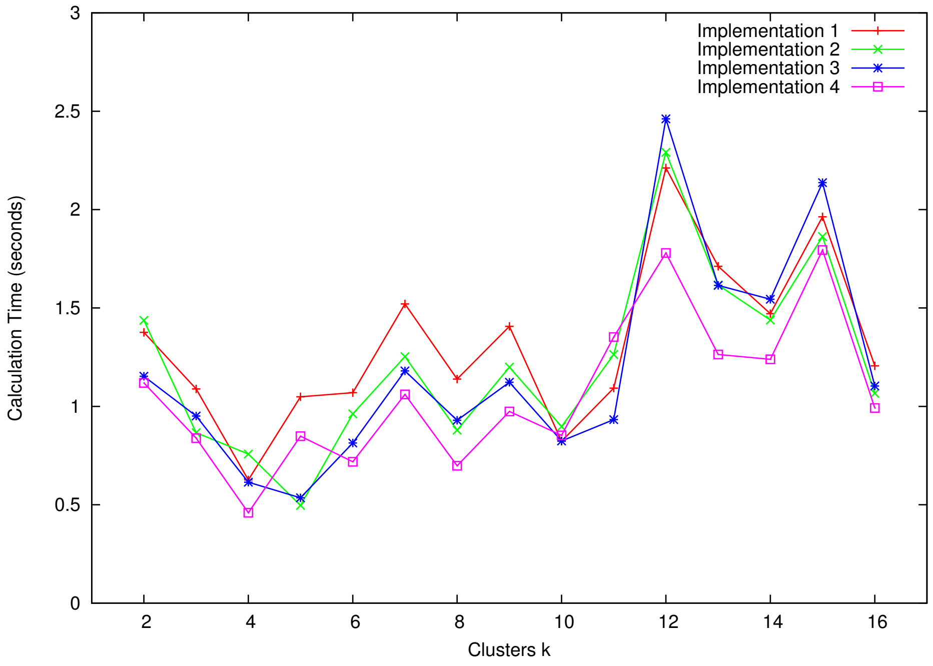


Figure 4: The calculation time of Implementation 1 to 4 for different k, using a convergence delta of 0.0001.

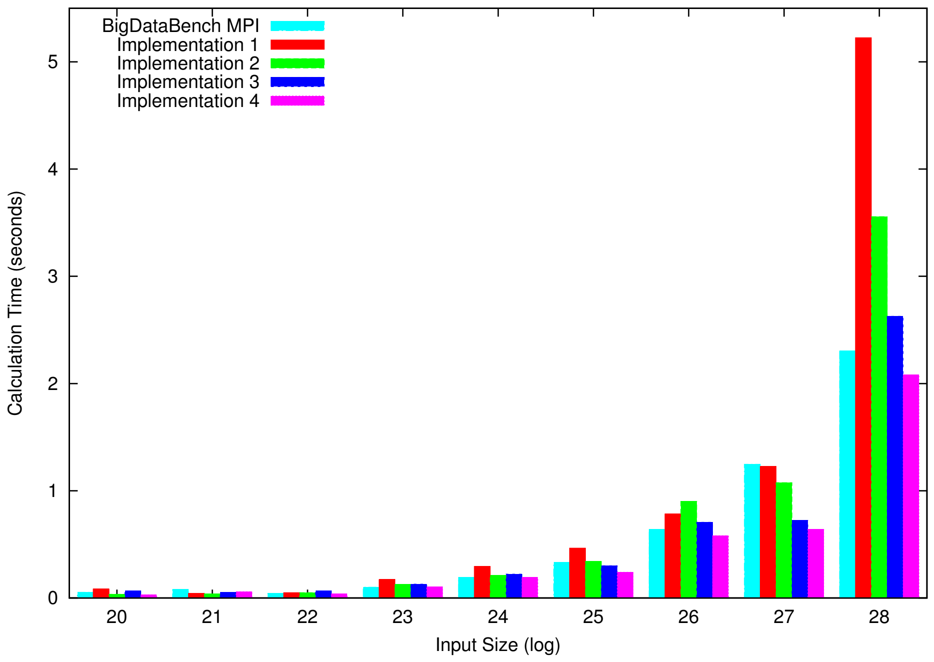


Figure 5: The calculation time of Implementation 1 to 4 and BigDataBench MPI on the 8×8 configuration, using threshold 0.0001.

formance, given that Implementation 4 is the best performing implementation in almost all cases. To get a better understanding of the performance of Implementation 1 to 4, we will use the BigDataBench MPI and Hadoop implementation as baselines.

For the comparison with the BigDataBench MPI implementation, we note that to allow a fair comparison we will measure only the time taken by the core calculation, similar to how we measure the time for Implementation 1 to 4 as noted before. The time measurement is taken from the moment the BigDataBench MPI implementation calls the function that will execute the iterations, until the moment the process has converged.

The performance of Implementation 1 to 4 and the BigDataBench MPI implementation on data sets of different sizes are shown in Figure 5. A graph showing the speedup (or speeddown) of Implementation 1 to 4 with respect to the BigDataBench MPI implementation is depicted in Figure 6. It should be noted that switching to a threshold of 0.0001 instead of a convergence delta caused outliers. For about 15% of the runs the number of iterations used became far greater then normally seen (up to 490 iterations in a single run, where runs with 3 to 10 iterations were normal). The BigDataBench MPI implementation also exhibited this behaviour. These outliers were excluded from the results shown.

The results show that the performance of Implementation 1 to 4 is close to the performance of the BigDataBench MPI implementation. Implementation 1, the slowest of the four Forelem implementations, proved to be slower than the BigDataBench MPI implementation. The fastest Forelem implementation, Implementation 4, proved to be faster for the majority of input sizes. Note that even though we take the average over 10 runs for each data point, the results still show some random fluctuation. For input sizes 2^{21} and 2^{27} this random fluctuation causes the BigDataBench MPI implementation to be slower than expected, therefore Figure 6 shows a slight elevation in speedup for these input sizes.

Finally, a comparison with the Hadoop implementation was made. Figure 7 shows the execu-

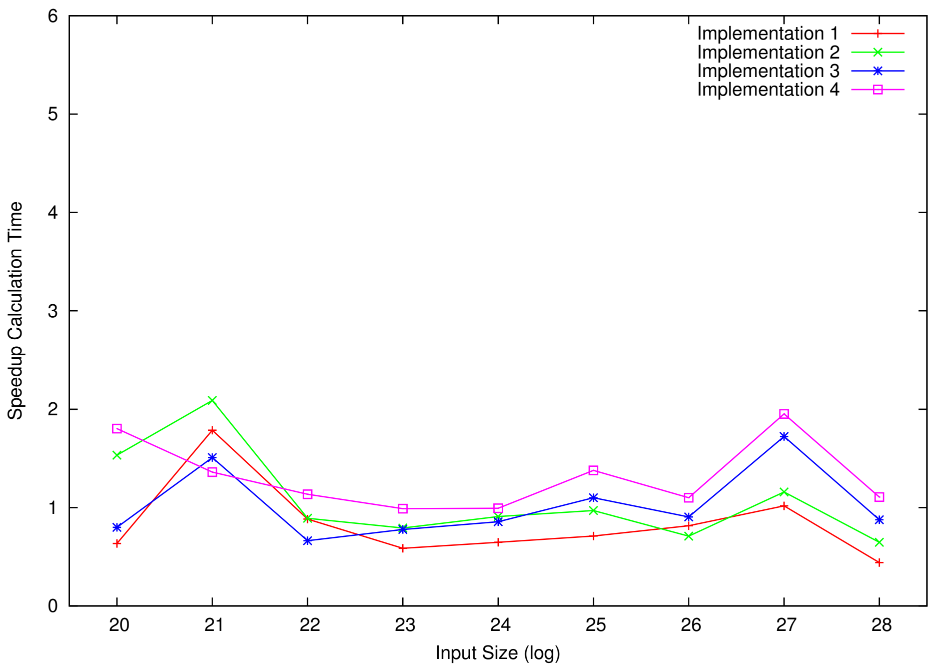


Figure 6: The speedup of Implementation 1 to 4 compared to BigDataBench MPI on the 8×8 configuration, using threshold 0.0001.

tion times of Implementation 1 to 4 and Figure 8 shows the execution times of the Hadoop implementation for different input sizes. Figure 9 shows the speedup of Implementation 1 to 4 compared to the Hadoop implementation for various input sizes. The Hadoop implementation was given a maximum number of iterations of 10. Note that Implementation 1 to 4 are between 20 to 70 times faster than the Hadoop implementation. While the Hadoop implementation first becomes more efficient compared to the Forelem implementations as the data size increases, it appears it then becomes less efficient for even larger data sizes. However, since for most implementations this effect is only shown for the largest data set, this may be coincidental (due to the randomness of the initialization and its influence on performance). It was not possible to run the Hadoop implementation for a data set larger than 2^{25} data points, because it runs out of memory.
Note that since the input and output operations of the Hadoop implementation are an inherent part of the implementation, we compare the Hadoop implementation with the Forelem implementations based on the overall execution time. Although the Hadoop implementation of k-means is compute intensive, we still have to take into account that the I/O overhead may be extensive, skewing the results in our favor. However, given the large difference in performance, we still conclude that the performance of the Forelem implementations is superior to the Hadoop implementation.

7 Conclusion

The Forelem framework can be used to derive implementations for k-means clustering. The application of a sequence of transformations to a simple specification of the algorithm leads to the derivation of highly efficient implementations. The performance of these implementations is shown to be superior to the performance of the Hadoop implementation provided by the

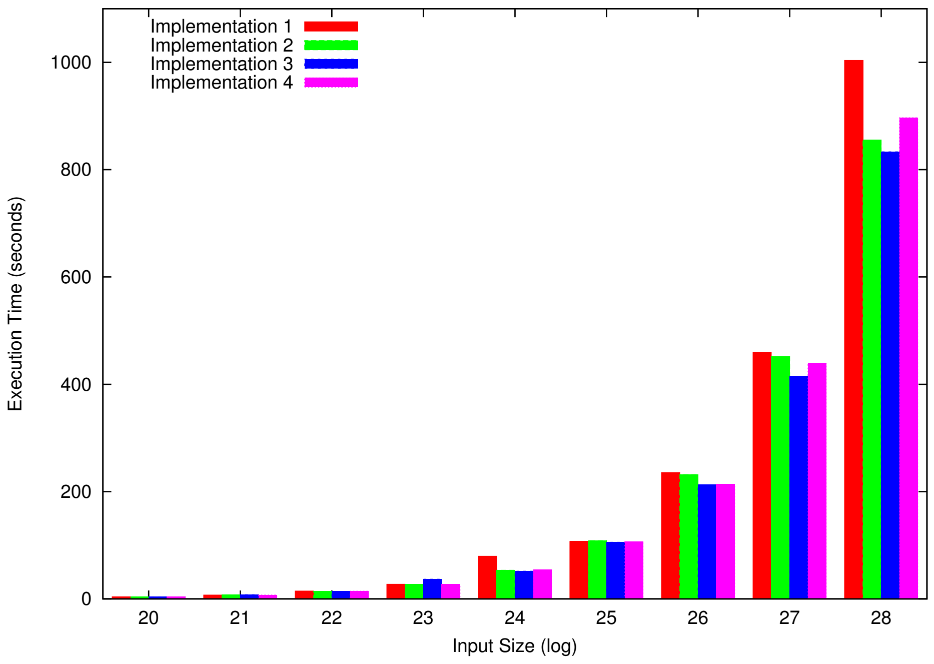


Figure 7: The execution time of Implementation 1 to 4 on the 8×8 configuration, using a convergence delta of 0.0001.

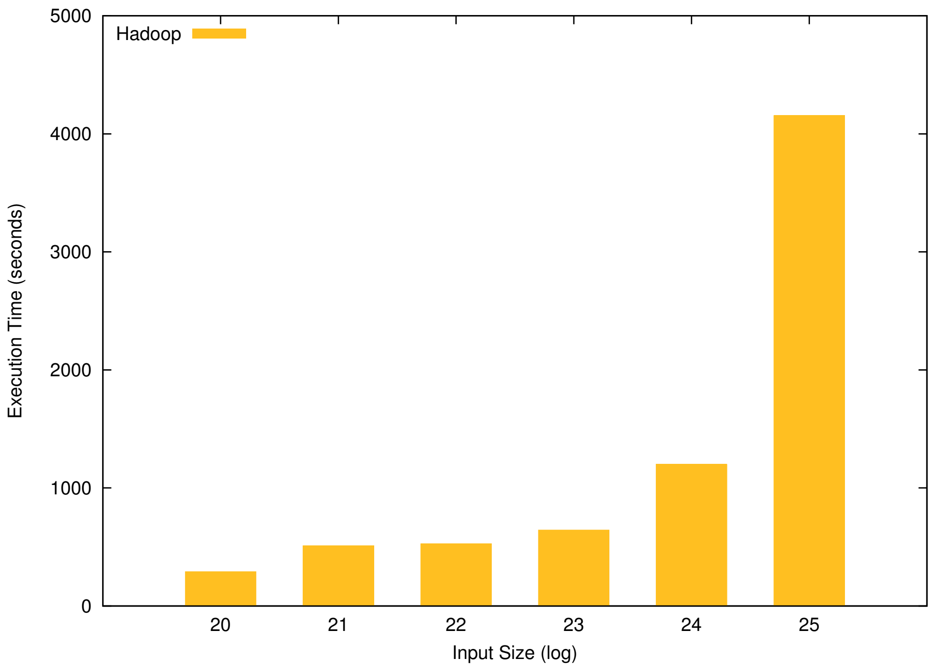


Figure 8: The execution time of Hadoop, using a convergence delta of 0.0001.

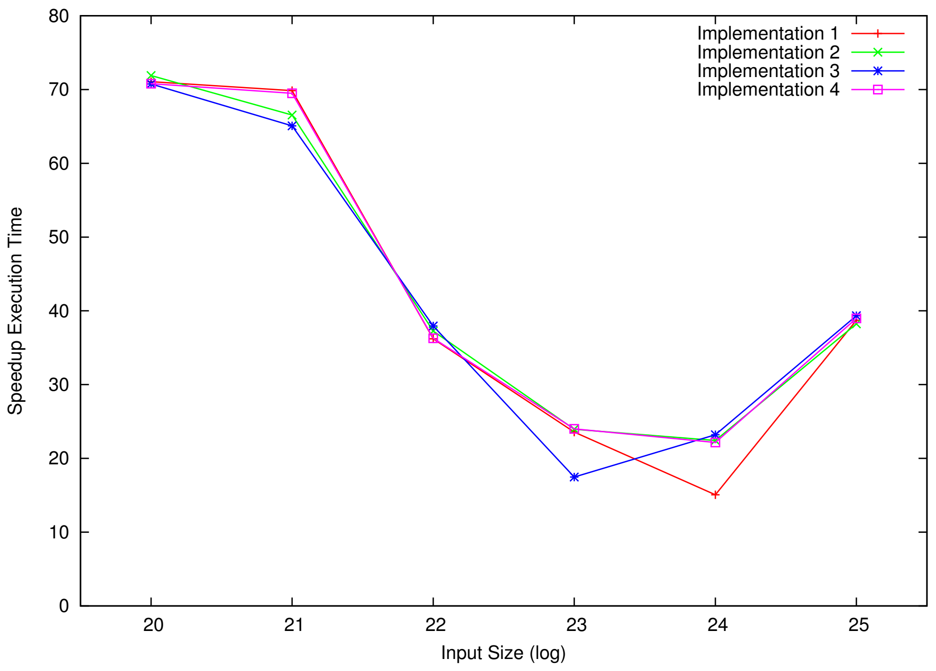


Figure 9: The speedup of Implementation 1 to 4 compared to Hadoop on the 8×8 configuration, using a convergence delta of 0.0001.

BigDataBench benchmark, being approximately 40 times faster. The most optimized implementation derived for the experiments is faster than the MPI implementation provided by the BigDataBench benchmark.
The Forelem framework allows the creation of inherently parallel specifications. This ensures that the specification must be created by reducing an existing algorithm to its core idea, and leads to implementations which scale well when increasing the number of threads. Reducing an algorithm to its core idea also allows the Forelem framework to generate codes using different communication schemes mechanically.
The implementations used for this research were derived from the initial specification, but ultimately were still programmed by hand. Future work will include the automation of this process, and to demonstrate the effectiveness of this framework on various other examples and algorithms.

References

[1] Apache mahout: Scalable machine learning and data mining. `http://mahout.apache.org`. Accessed: 2017-07-14.

[2] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[3] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[4] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[5] A. K. Jain. Data clustering: 50 years beyond k-means. In *Pattern Recognition Letters*, volume 31, pages 651–666, 2010.

[6] C. Luo Y. Zhu Q. Yang Y. He W. Gao Z. Jia Y. Shi S. Zhang C. Zheng G. Lu K. Zhan X. Li L. Wang, J. Zhan and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[7] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. Automata, Languages and Programming (Lecture Notes in Computer Science)*, volume 115, 1981.

[8] W.-K. Liao. Paralel k-means data clustering. `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`. Accessed: 2017-07-22.

[9] K. F. D. Rietveld and H. A. G. Wijshoff. Forelem: A Versatile Optimization Framework For Tuple-Based Computations. In *CPC 2013: 17th Workshop on Compilers for Parallel Computing*, July 2013.

[10] K. F. D. Rietveld and H. A. G. Wijshoff. Towards a new tuple-based programming paradigm for expressing and optimizing irregular parallel computations. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 16:1–16:10, 2014.

[11] K. F. D. Rietveld and H. A. G. Wijshoff. Reducing layered database applications to their essence through vertical integration. *ACM Trans. Database Syst.*, 40(3):18:1–18:39, 2015.

[12] K. F. D. Rietveld and H. A. G. Wijshoff. Optimizing sparse matrix computations through compiler-assisted programming. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 100–109, 2016.

[13] H. Steinhaus. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci. Cl. III. 4*, pages 801–804, 1956.

[14] B. van Strien, K. F. D. Rietveld, and H. A. G. Wijshoff. Deriving highly efficient implementations of parallel pagerank. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017.