Parallelising K-means

Harry Scells - https://github.com/hscells/k-means - N8857580

October 2015

Contents

1 Defining K-means **2**
1.1 Sequential algorithm 2
1.2 Parallel algorithm 3

2 Challenges Involved **4**
2.1 Why parallelising k-means is difficult 4
2.2 Changes required to expose parallelism 4

3 Computation Mapping **5**
3.1 Sequential Call Graph 5
3.2 Parallel Call Graph 5
3.3 Mapping computation to processors 6

4 Timing & Profiling Results **6**
4.1 Profiling Results 7
4.2 Explanation 8

5 Tools Used **9**

6 Overcoming Performance Problems **9**

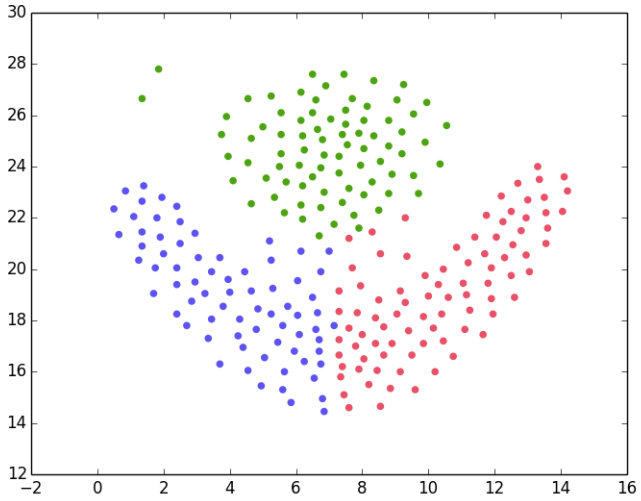
7 Source Code **10**

8 Reflection **11**

9 Appendix **12**

1 Defining K-means

K-means is a method of vector quantisation often used in data mining. The algorithm is very good at clustering very large datasets and is reasonable at doing this with high groupings.



The points above represent the two dimensional observation set, clustered where k=3. Traditionally, k-means will produce roughly equal clusters, and can be compared to a Voronoi diagram.

1.1 Sequential algorithm

The k-means algorithm is implemented using k clusters over an n sized dataset. Traditionally, k-means can be used to classify data over an n dimensional dataset, however for simplicity the implementation that this report covers is two-dimensional. Sequentially, k-means aims to partition n two dimensional vectors into k sets. This can be expressed formally as:

$$\sum_{i=0}^{n} \min_{\mu_i \epsilon C}(||x_i - \mu_i||)^2$$

The algorithm partitions a set of n observations into k disjoint clusters C, each described by the mean μ_i. In the implementation using clojure, this is done using two dependent functions: one which groups the set of observations into one of k sets and the other which takes the new groups and shifts the centroids to the mean of each of the new groupings. This process is done as many times as needed until equilibrium occurs. This is determined by testing to see if the centroids have stopped moving.

1.2 Parallel algorithm

While this may seem trivial, k-means is an NP-hard problem. Due to the fact that Clojure is a functional language, the notion of mutating variables does not need to be taken into account. Both implementations are stateless, functional programs. Profiling and analysing the code becomes very simple, and rather than taking one independent section of code and modifying that, small parts of the program can be performed concurrently. So long as functions take the same input parameters, returns some value, and has no side effects, the underlying code can be changed without worrying about mutating anything outside the scope of that function. This is a benefit of writing parallel programs with "pure" functions. This is how the sequential code has been parallelised; by taking existing sequential code and making granular changes to individual functions.

The approach taken in this project was to take the set of observations and partition it into t partitions, where t is the number of cores. These subsets are then computed concurrently for each centroid and then joined at the end. The only slow part is the re-grouping of the observations.

```
1 (let
2       [futures (doall (map #(future (cluster % c)) (partition-all (int
            (/ (count l) n-cpu)) l)))
3        clusters (map deref futures)]
4       (group-clusters clusters))
```

The code above demonstrates the pattern used for calling the sequential code concurrently. A map of futures are applied to the partitioned set of clusters, these futures work on calling the sequential cluster code, "cluster" and another map is used to observe the value which has been computed. It should be noted that dereferencing the futures blocks until it's finished. Finally, the multiple sets of clusters are joined into one.

Another way to tackle this problem would be to give each centroid a copy of the entire set of observations to work on and then reduce the lists, assigning individual observations to the centroid which computes the maximum value for that point. Intuitively this appears to have some downsides: giving each centroid a copy of the list is an expensive memory operation, very large datasets and very small k values result in each thread doing more work than the alternative solution, and finally there is still the problem of having to join or reduce at the end of the computations.

2 Challenges Involved

2.1 Why parallelising k-means is difficult

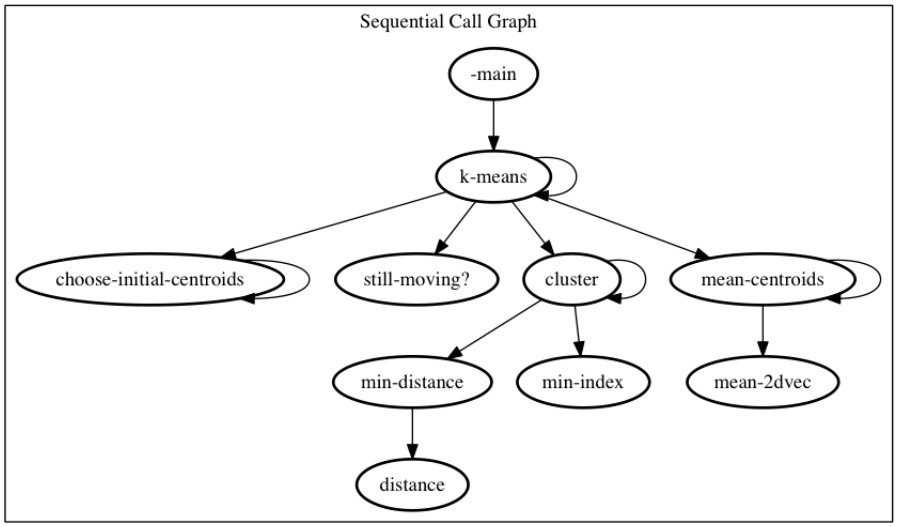
The underlying process of k-means can't be changed due to the fact that clustering the observations to the centroids depends on the shifting of the centroids to the observations. However, the individual functions that perform these operations can be modified to employ concurrent paradigms. K-means is an NP-hard problem which means a nondeterministic solution exists for it which can be solved in an unknown polynomial time. Given a set of n points, the resulting set of clusters will be different every time. There are no clustering algorithms specifically designed to cluster in parallel. Different clustering algorithms are also used for different circumstances. For these reasons most clustering algorithms have a sequential and parallel counterpart.

2.2 Changes required to expose parallelism

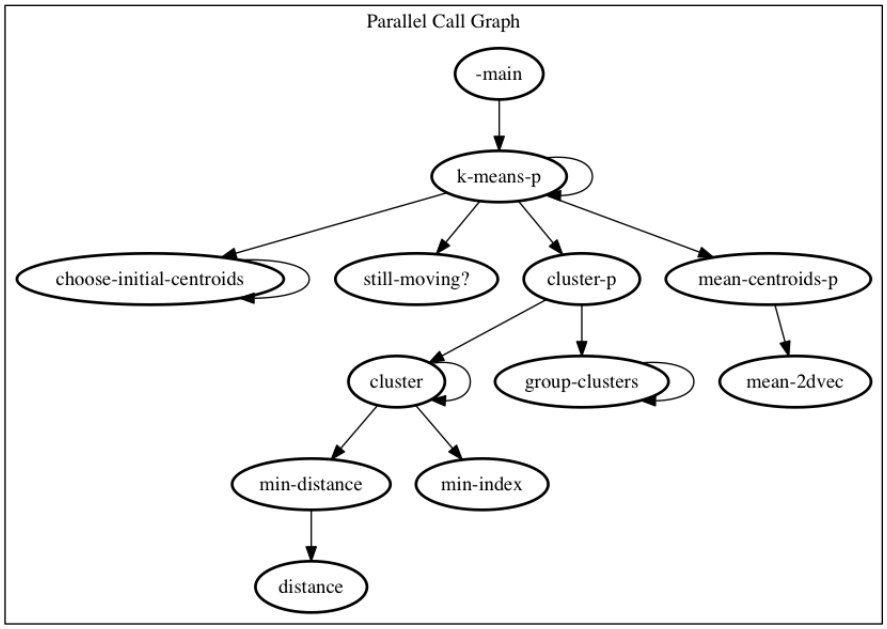
While the process cannot be changed, it can be parallelised to an extent. This was done by partitioning the set of observations into t groups and performing the same sequential clustering function on each subset. Each subset is then joined back into one set containing k subsets (where t is the number of threads and k is the number of centroids). This code is done concurrently using future constructs. for each partition created, a future is also created and then blocks until the answer becomes realised.

3 Computation Mapping

3.1 Sequential Call Graph



3.2 Parallel Call Graph



3.3 Mapping computation to processors

As can be seen in the above call graphs, the parallel clustering code makes calls to sequential code. The set of observations is split as evenly as possible into t chunks. Each partition has the sequential algorithm run on it, and once all threads are finished, the t sets are joined into one set of k observations. To make this clear, rather than performing one large calculation over the entire set, because each observation is an independent 'event' it doesn't matter which order the observations are calculated in. The parallel abstraction used to achieve this is called a future (or promise or delay). The function and data is sent to a thread pool and the calling function will block until all futures are realised. That is, until the computation of the future is finished, and some data is available to return.

4 Timing & Profiling Results

All of the runs listed below were performed on the QUT bigdata computing cluster. Each run is performed on a random dataset of 500,000 observations, the k value for each run was also set at 5. Each run is also non-deterministic in the sense that for each run there is a maximum iteration cap of 100 iterations of the algorithm, but potentially the program can stop before that point. This is simply the nature of k-means. The parallel code was run using a t value of 8 (the maximum processors).

4.1 Profiling Results

The results for each run are displayed below:

Sequential Run #1 (43 iterations)

Id	nCalls	Min	Max	Mean	Time%	Time
cluster	43	5.7s	9.3s	5.8s	85	214.6s
mean-centroids	43	1.0	1.2s	1.1s	15	45.9s
mean-2dvec	215	130.0ms	330.0ms	213.0ms	15	45.9s
min-distance	21,500,000	75ns	1.0ms	101ns	1	1.9s
choose-initial-centroids	1	53.0ms	53.0ms	53.0ms	0	53.0ms
Clock Time					100	300.6s
Accounted Time					122	367.4s

Sequential Run #2 (37 iterations)

Id	nCalls	Min	Max	Mean	Time%	Time
cluster	37	5.6s	8.9s	5.8s	85	214.6s
mean-centroids	37	1.0s	1.2s	1.0s	15	31.7s
mean-2dvec	185	148.0ms	324.0ms	209.0ms	15	38.7s
min-distance	18,500,000	68ns	1.0ms	101ns	1	1.9s
choose-initial-centroids	1	62.0ms	62.0ms	62.0ms	0	62.0ms
Clock Time					100	253.7s
Accounted Time					122	309.7s

Parallel Run #1 (43 iterations)

Id	nCalls	Min	Max	Mean	Time%	Time
cluster-parallel	43	3.5s	4.1s	3.6s	92	156.9s
group-clusters	43	3.4s	4.0s	3.6s	90	153.5s
mean-2dvec	215	79.0ms	526.0ms	228.0ms	29	49.0s
min-distance	21,500,000	114ns	26.0ms	209ms	3	4.5s
choose-initial-centroids	1	47.0ms	47.0ms	47.0ms	0	47.0ms
mean-centroids-parallel	43	3.0µs	1.0ms	35.0µs	0	1.0ms
Clock Time					100	170.4s
Accounted Time					260	443.4s

Parallel Run #2 (42 iterations)

Id	nCalls	Min	Max	Mean	Time%	Time
cluster-parallel	42	3.4s	4.2s	3.6s	94	152.7s
group-clusters	42	3.3s	4.1s	3.6s	92	149.6s
mean-2dvec	210	171.0ms	349.0ms	219.0ms	28	46.0s
min-distance	21,000,000	108ns	10.0ms	199ns	3	4.2s
choose-initial-centroids	1	42.0ms	42.0ms	42.0ms	0	42.0ms
mean-centroids-parallel	42	3.0µs	1.0ms	33.0µs	0	1.0ms
Clock Time					100	162.5s
Accounted Time					264	428.0s

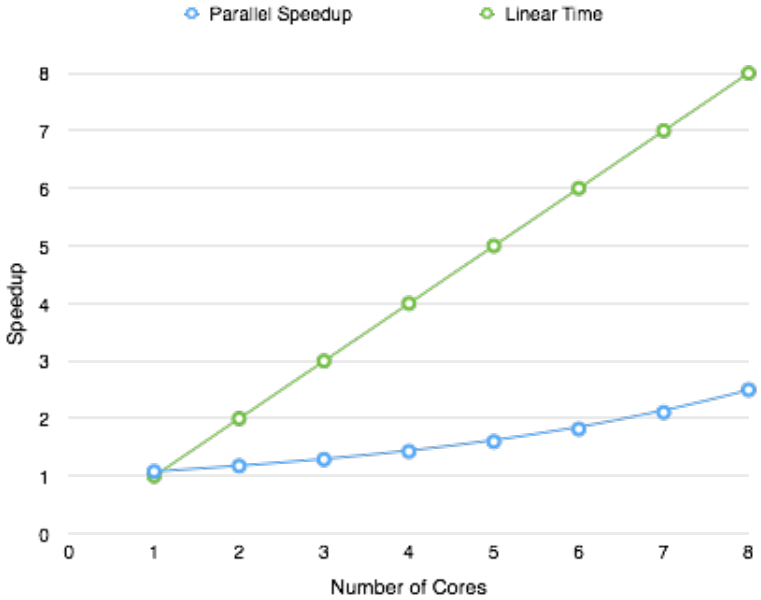
4.2 Explanation

Both the sequential and parallel variant have the same functions, except for group-clusters, which is unique to the parallel implementation. The parallel variants of the functions used in the sequential code are denoted here with the "-parallel" suffix, and in the code with the "-p" suffix. The biggest improvement overall came from the mean-centroids function, which came down from |40| seconds overall to |1| millisecond. Surprisingly, calling the sequential cluster function concurrently actually runs faster than simply running the sequential code on it's own. This goes to show that distributing data across many processes can provide a very good speedup when compared to only using one process to process all data at once.

The speedup of the parallel code is calculated in the following way:

$$Speedup = \frac{(300.6 + 253.7)/2}{(170.4 + 162.5)/2} = \frac{427.25}{166.45} = 2.56$$

It can be seen that through parallelisation, a speedup of 2.56 times can be achieved. The best case performance of the sequential algorithm is $O(ikn)$, parallelising the algorithm, however, results in a complexity of $O((i/t)kn)$. The i value gets divided by t, because the vector of observations is split into k partitions for each thread to work on.



This graph shows the speedup obtained from the parallel algorithm. It can be seen that, like the calculation above, a speedup of 2.5 can be achieved when using up to eight cores. While a small number gives barely any performance improvements, it can be predicted that by adding more processors, the speedup will be better than linear time eventually.

5 Tools Used

The algorithm was programmed in Clojure with help from the Leiningen project automator. Also used were:

- timbr for profiling and timing statistics.
- the numpy and matplotlib python libraries for visualising the output of the program.

The timbr profiling library is the best Clojure profiling tool. While the program could be analysed using a Java profiler, this was a realisation too late. Timbr provides a decent amount of information about the program anyway, enough to gain insight about the performance of it.

6 Overcoming Performance Problems

Due to the fact that k-means is an NP-hard problem, there are only very few ways to make the algorithm better. Many attempts were made at increasing the performance, however, due to the fact than most parts of the algorithm rely on the completion of a previous section of code, there are not too many places for improvement. Clojure comes with a variety of very good list-comprehension functions built-in, and this made things like partitioning a list of elements and performing list operations like mappings much easier than traditional imperative languages. Evenly splitting the list in a language like C for instance, would involve writing a performance-critical list splitting function. In Clojure, however, there was already a function that does this. Using a Lisp-based language provided an instantaneous benefit for large lists of data and transforming that data. There were downsides to this however; naively in the first parallel implementation, all map functions were replaced with pmap functions. The parallel map function internally creates a new thread for each element in the list, so attempting to implicitly spawn thousands of threads at once proved less than ideal. This was because the overhead involved in spawning the thread pool, attaching and detaching threads was longer than it took for the actual computation involved. On the computation mapping side of things, one attempt had the parallel code try to cluster the set of observations for each centroid. This was a bad idea for two reasons: it meant that the parallel code was doing k times as much work, and the work involved in getting the smallest value out of k values is also more work than simply joining k lists. Clojure is one of the fastest JVM (Java Virtual Machine) languages, and is used heavily for concurrent code in the industry, however the results obtained here should not be comparable to the same algorithm written in C (Although this was never tested).

7 Source Code

The sequential source code for the function which performs the k-means is below. The code for the parallel program is exactly the same except it calls parallel versions of cluster/cluster and cluster/mean-centroids. The parallel versions of those functions are denoted with the suffix p

```
(defn k-means
  "Perform the actual k-means"
  ([l k max-iterations]
    (k-means l k (choose-initial-centroids l k) (vector/make-vector k)
        max-iterations))
  ([l k c g max-iterations]
    (let [t c g (cluster/cluster l c) c (cluster/mean-centroids g)]
      (println "Iteration" (- 100 max-iterations) c)
      (cond
        (> 0 max-iterations) g
        (still-moving? t c) ; Keep shifting centroids until equilibrium
          (recur l k c g (dec max-iterations))
        :else g))))
```

The sequential program uses the below algorithm to compute a set of observations into k sets of observations.

```
(defn cluster
  "take a list of vectors and cluster on centroids c via distance"
  ([l c] (cluster l c (vector/make-vector (count c))))
  ([l c a]
    (cond
      (empty? l) a
      ; otherwise we continue until the list of vectors is empty
      (not-empty l)
        ; for the vector at the start of the list, choose a centroid
            which is closest
        (let [i (min-index (min-distance c (first l)))]
          ; append the vector to the smallest distance
          (recur (rest l) c (assoc a i (conj (nth a i) (first l)))))
      :else
      a)))
```

The parallel program uses the same algorithm as above, however, it partitions the set of observations to be computes in parallel.

```
(defn cluster-p
  "take a list of vectors and cluster on centroids c via distance in
      parallel"
  [^ints l ^floats c]
    ; first split the list of observations into chunks and run the
        sequental cluster algorithm over each chunk
    (let [futures (doall (map #(future (cluster % c)) (partition-all (
        int (/ (count l) n-cpu)) l))) clusters (map deref futures)]
      ; secondly, join each sub-set of clusters into one large set
      (group-clusters clusters)))
```

The shifting of the centroids is performed by mean-centroids and mean-centroids-p, and both functions both call to mean-2dvec. The conversion to parallel code was made very simple and it ended up being even less code than the sequential implementation.

```
1  (defn mean-2dvec
2    [^ints v]
3    (cond
4      (empty? v) v
5      :else
6        (let [
7          a (map first v)
8          b (flatten (map rest v))
9          x (float (/ (reduce + a) (count a)))
10         y (float (/ (reduce + b) (count b)))]
11         (list x y))))
12
13 (defn mean-centroids
14   "Calculates the mean for a given centroid"
15   ([g] (mean-centroids g []))
16   ([g l]
17     (cond
18       (empty? g) l
19       :else
20         (recur (rest g) (conj l (mean-2dvec (first g)))))))
21
22 (defn mean-centroids-p
23   "Calculates the mean for a given centroid in parallel"
24   [g]
25   (pmap (mean-2dvec %) g))
```

8 Reflection

The outcome of this project is clear - a speedup of a considerable amount was achieved on an NP-Hard problem. Using a functional language which is based on LISP helped due to the fact that Clojure treats code as data. A list-processing language makes abstracting and visualising the problem much easier than an object-oriented imperative language. By having these great abstractions available it was possible to design the concurrent algorithm as a modular solution. The stateless nature of Clojure and it's first-class functions mean that code reuse is high, and code which is inherently sequential can be made parallel by looking for data dependencies. By modeling observations and assuming that each point is independent, it doesn't matter in which order the set of all observations are computed.

K-means has had much research into making it parallel, and the work that has been done here works well. However, there are several assumptions which have been made, in particular: all observations are independent, and these observations are present in two-dimensional space. For higher dimensional observation spaces, this algorithm is neither suited to compute a result, or feasible. Computing k observations in d dimensional space would have been too much for a semesters worth of work and would have been massively more difficult to parallelise.

Given more time, it would have been very interesting in modifying the code to allow for calculations of d dimensional spaces, but it would most likely involve

a total rewrite of the algorithm. The current algorithm however, is good at clustering very large sets of data in an acceptable space of time, given the JVM. If something were to be done differently, it would be to use a functional language which is compiled, rather than run in a virtual machine. This however, was a choice that was made early on in the course and the downsides of having concurrent code running in a virtual machine does not come as a surprise in terms of performance.

9 Appendix

- Documentation for the project http://hscells.github.io/k-means/doc/
- Source code https://github.com/hscells/k-means
- Binary https://github.com/hscells/k-means/releases/tag/1.0